

utils.py

```
"""Utilities to create and set the config.

The main class exported by this module is :class:`ManimConfig`. This class
contains all configuration options, including frame geometry (e.g. frame
height/width, frame rate), output (e.g. directories, logging), styling
(e.g. background color, transparency), and general behavior (e.g. writing a
movie vs writing a single frame).

See :doc:`/guides/configuration` for an introduction to Manim's configuration system.

"""

from __future__ import annotations

import argparse
import configparser
import copy
import errno
import logging
import os
import re
import sys
from collections.abc import Mapping, MutableMapping
from pathlib import Path
from typing import Any, Iterable, Iterator

import colour
import numpy as np

from .. import constants
from ..constants import RendererType
from ..utils.tex import TexTemplate, TexTemplateFromFile
from ..utils.tex_templates import TexTemplateLibrary

def config_file_paths() -> list[Path]:
    """The paths where ``.cfg`` files will be searched for.

    When manim is first imported, it processes any ``.cfg`` files it finds. This
    function returns the locations in which these files are searched for. In
    ascending order of precedence, these are: the library-wide config file, the
    user-wide config file, and the folder-wide config file.

    The library-wide config file determines manim's default behavior. The
    user-wide config file is stored in the user's home folder, and determines
    the behavior of manim whenever the user invokes it from anywhere in the
    system. The folder-wide config file only affects scenes that are in the
    same folder. The latter two files are optional.

    These files, if they exist, are meant to loaded into a single
    :class:`configparser.ConfigParser` object, and then processed by
    :class:`ManimConfig`.

    Returns
    ------
    List[:class:`Path`]
        List of paths which may contain ``.cfg`` files, in ascending order of
        precedence.

    See Also
    -----
    :func:`make_config_parser`, :meth:`ManimConfig.digest_file`,
    :meth:`ManimConfig.digest_parser`

    Notes
    -----
    The location of the user-wide config file is OS-specific.

    """
    library_wide = Path.resolve(Path(__file__).parent / "default.cfg")
    if sys.platform.startswith("win32"):
        user_wide = Path.home() / "AppData" / "Roaming" / "Manim" / "manim.cfg"
    else:
        user_wide = Path.home() / ".config" / "manim" / "manim.cfg"
    folder_wide = Path("manim.cfg")
    return [library_wide, user_wide, folder_wide]

def make_config_parser(
    custom_file: str | os.PathLike | None = None,
) -> configparser.ConfigParser:
    """Make a :class:`ConfigParser` object and load any ``.cfg`` files.

    The user-wide file, if it exists, overrides the library-wide file. The
    folder-wide file, if it exists, overrides the other two.

    The folder-wide file can be ignored by passing ``custom_file``. However,
    the user-wide and library-wide config files cannot be ignored.

    Parameters
    -----
    custom_file
        Path to a custom config file. If used, the folder-wide file in the
        relevant directory will be ignored, if it exists. If None, the
        folder-wide file will be used, if it exists.

    Returns
    ------
    :class:`ConfigParser`
        A parser containing the config options found in the .cfg files that
        were found. It is guaranteed to contain at least the config options
        found in the library-wide file.

    See Also
    -----
    :func:`config_file_paths`

    """
    library_wide, user_wide, folder_wide = config_file_paths()
    # From the documentation: "An application which requires initial values to
    # be loaded from a file should load the required file or files using
    # read_file() before calling read() for any optional files."
    # https://docs.python.org/3/library/configparser.html#configparser.ConfigParser.read
    parser = configparser.ConfigParser()
    with library_wide.open() as file:
        parser.read_file(file) # necessary file
    other_files = [user_wide, Path(custom_file) if custom_file else folder_wide]
    parser.read(other_files) # optional files

    return parser

def determine_quality(qual: str) -> str:
    for quality, values in constants.QUALITIES.items():
        if values["flag"] is not None and values["flag"] == qual:
            return quality

    return qual
```

```
class ManimConfig(MutableMapping):
    """Dict-like class storing all config options.

    The global ``config`` object is an instance of this class, and acts as a single source of truth for all of the library's customizable behavior.

    The global ``config`` object is capable of digesting different types of sources and converting them into a uniform interface. These sources are (in ascending order of precedence): configuration files, command line arguments, and programmatic changes. Regardless of how the user chooses to set a config option, she can access its current value using :class:`ManimConfig`'s attributes and properties.
```

Notes

Each config option is implemented as a property of this class.

Each config option can be set via a config file, using the full name of the property. If a config option has an associated CLI flag, then the flag is equal to the full name of the property. Those that admit an alternative flag or no flag at all are documented in the individual property's docstring.

Examples

We use a copy of the global configuration object in the following examples for the sake of demonstration; you can skip these lines and just import ``config`` directly if you actually want to modify the configuration:

.. code-block:: pycon

```
>>> from manim import config
>>> config = global_config.copy()
```

Each config option allows for dict syntax and attribute syntax. For example, the following two lines are equivalent,

.. code-block:: pycon

```
>>> from manim import WHITE
>>> config.background_color = WHITE
>>> config["background_color"] = WHITE
```

The former is preferred; the latter is provided mostly for backwards compatibility.

The config options are designed to keep internal consistency. For example, setting ``frame_y_radius`` will affect ``frame_height``:

.. code-block:: pycon

```
>>> config.frame_height
8.0
>>> config.frame_y_radius = 5.0
>>> config.frame_height
10.0
```

There are many ways of interacting with config options. Take for example the config option ``background_color``. There are three ways to change it: via a config file, via CLI flags, or programmatically.

To set the background color via a config file, save the following ``manim.cfg`` file with the following contents.

.. code-block:::

```
[CLI]
background_color = WHITE
```

In order to have this ``.cfg`` file apply to a manim scene, it needs to be placed in the same directory as the script,

.. code-block:: bash

```
project/
  "x"Scene.py
  "x"manim.cfg
```

Now, when the user executes

.. code-block:: bash

```
manim scene.py
```

the background of the scene will be set to ``WHITE``. This applies regardless of where the manim command is invoked from.

Command line arguments override ``.cfg`` files. In the previous example, executing

.. code-block:: bash

```
manim scene.py -c BLUE
```

will set the background color to BLUE, regardless of the contents of ``manim.cfg``.

Finally, any programmatic changes made within the scene script itself will override the command line arguments. For example, if ``scene.py`` contains the following

.. code-block:: python

```
from manim import *
config.background_color = RED
```

```
class MyScene(Scene):
    ...
```

the background color will be set to RED, regardless of the contents of ``manim.cfg`` or the CLI arguments used when invoking manim.

"""

```
_OPTS = {
    "assets_dir",
    "background_color",
    "background_opacity",
    "custom_folders",
    "disable_caching",
    "disable_caching_warning",
    "dry_run",
    "enable_wireframe",
    "ffmpeg_loglevel",
    "ffmpeg_executable",
    "format",
    "flush_cache",
    "frame_height",
    "frame_rate",
    "frame_width",
    "frame_x_radius",
```

```

"frame_y_radius",
"from_animation_number",
"images_dir",
"input_file",
"media_embed",
"media_width",
"log_dir",
"log_to_file",
"max_files_cached",
"media_dir",
"movie_file_extension",
"notify_outdated_version",
"output_file",
"partial_movie_dir",
"pixel_height",
"pixel_width",
"plugins",
"preview",
"progress_bar",
"quality",
"save_as_gif",
"save_sections",
"save_last_frame",
"save_pngs",
"scene_names",
"show_in_file_browser",
"tex_dir",
"tex_template",
"tex_template_file",
"text_dir",
"upto_animation_number",
"renderer",
"enable_gui",
"gui_location",
"use_projection_fill_shaders",
"use_projection_stroke_shaders",
"verbosity",
"video_dir",
"sections_dir",
"fullscreen",
>window_position",
>window_size",
>window_monitor",
"write_all",
"write_to_movie",
"zero_pad",
"force_window",
}

def __init__(self) -> None:
    self._d = {k: None for k in self._OPTS}

# behave like a dict
def __iter__(self) -> Iterator[str]:
    return iter(self._d)

def __len__(self) -> int:
    return len(self._d)

def __contains__(self, key) -> bool:
    try:
        self.__getitem__(key)
        return True
    except AttributeError:
        return False

def __getitem__(self, key) -> Any:
    return getattr(self, key)

def __setitem__(self, key: str, val: Any) -> None:
    setattr(ManimConfig, key).fset(self, val) # fset is the property's setter

def update(self, obj: ManimConfig | dict) -> None:
    """Digest the options found in another :class:`ManimConfig` or in a dict.

    Similar to :meth:`dict.update`, replaces the values of this object with those of ``obj``.

    Parameters
    -----
    obj
        The object to copy values from.

    Returns
    -----
    None

    Raises
    -----
    :class:`AttributeError`
        If ``obj`` is a dict but contains keys that do not belong to any config options.

    See Also
    -----
    :meth:`ManimConfig.digest_file`, :meth:`~ManimConfig.digest_args`,
    :meth:`ManimConfig.digest_parser`
    """
    if isinstance(obj, ManimConfig):
        self._d.update(obj._d)
        if obj.tex_template:
            self.tex_template = obj.tex_template

    elif isinstance(obj, dict):
        # First update the underlying _d, then update other properties
        _dict = {k: v for k, v in obj.items() if k in self._d}
        for k, v in _dict.items():
            self[k] = v

        _dict = {k: v for k, v in obj.items() if k not in self._d}
        for k, v in _dict.items():
            self[k] = v

    # don't allow to delete anything
    def __delitem__(self, key: str):
        raise AttributeError("ManimConfig object does not support item deletion")

    def __delattr__(self, key: str):
        raise AttributeError("ManimConfig object does not support item deletion")

    # copy functions
    def copy(self) -> ManimConfig:
        """Deepcopy the contents of this ManimConfig.

        Returns
        -----
        :class:`ManimConfig`
            A copy of this object containing no shared references.

        See Also
        -----
        :func:`tempconfig`
    """

```

```

Notes
-----
This is the main mechanism behind :func:`tempconfig`.

"""
return copy.deepcopy(self)

def __copy__(self) -> ManimConfig:
    """See ManimConfig.copy()."""
    return copy.deepcopy(self)

def __deepcopy__(self, memo: dict[str, Any]) -> ManimConfig:
    """See ManimConfig.copy()."""
    c = ManimConfig()
    # Deepcopying the underlying dict is enough because all properties
    # either read directly from it or compute their value on the fly from
    # values read directly from it.
    c._d = copy.deepcopy(self._d, memo)
    return c

# helper type-checking methods
def _set_from_list(self, key: str, val: Any, values: list) -> None:
    """Set ``key`` to ``val`` if ``val`` is contained in ``values``."""
    if val in values:
        self._d[key] = val
    else:
        raise ValueError(f"attempted to set {key} to {val}; must be in {values}")

def _set_from_enum(self, key: str, enum_value: Any, enum_class: EnumMeta) -> None:
    """Set ``key`` to the enum object with value ``enum_value`` in the given
    ``enum_class``."""

Tests:::

>>> from enum import Enum
>>> class Fruit(Enum):
...     APPLE = 1
...     BANANA = 2
...     CANTALOUE = 3
>>> test_config = ManimConfig()
>>> test_config._set_from_enum("fruit", 1, Fruit)
>>> test_config._d['fruit']
<Fruit.APPLE: 1>
>>> test_config._set_from_enum("fruit", Fruit.BANANA, Fruit)
>>> test_config._d['fruit']
<Fruit.BANANA: 2>
>>> test_config._set_from_enum("fruit", 42, Fruit)
Traceback (most recent call last):
...
ValueError: 42 is not a valid Fruit
"""
self._d[key] = enum_class(enum_value)

def _set_boolean(self, key: str | int, val: Any) -> None:
    """Set ``key`` to ``val`` if ``val`` is Boolean."""
    if val in [True, False]:
        self._d[key] = val
    else:
        raise ValueError(f"{key} must be boolean")

def _set_tuple(self, key: str, val: tuple) -> None:
    if isinstance(val, tuple):
        self._d[key] = val
    else:
        raise ValueError(f"{key} must be tuple")

def _set_str(self, key: str, val: Any) -> None:
    """Set ``key`` to ``val`` if ``val`` is a string."""
    if isinstance(val, str):
        self._d[key] = val
    elif not val:
        self._d[key] = ""
    else:
        raise ValueError(f"{key} must be str or falsy value")

def _set_between(self, key: str, val: float, lo: float, hi: float) -> None:
    """Set ``key`` to ``val`` if lo <= val <= hi."""
    if lo <= val <= hi:
        self._d[key] = val
    else:
        raise ValueError(f"{key} must be {lo} <= {key} <= {hi}")

def _set_int_between(self, key: str, val: int, lo: int, hi: int) -> None:
    """Set ``key`` to ``val`` if lo <= val <= hi."""
    if lo <= val <= hi:
        self._d[key] = val
    else:
        raise ValueError(
            f"{key} must be an integer such that {lo} <= {key} <= {hi}",
        )

def _set_pos_number(self, key: str, val: int, allow_inf: bool) -> None:
    """Set ``key`` to ``val`` if ``val`` is a positive integer."""
    if isinstance(val, int) and val > -1:
        self._d[key] = val
    elif allow_inf and val in [-1, float("inf")]:
        self._d[key] = float("inf")
    else:
        raise ValueError(
            f"{key} must be a non-negative integer (use -1 for infinity)",
        )

def __repr__(self) -> str:
    rep = ""
    for k, v in sorted(self._d.items(), key=lambda x: x[0]):
        rep += f"(k: {v}, "
    return rep

# builders
def digest_parser(self, parser: configparser.ConfigParser) -> ManimConfig:
    """Process the config options present in a :class:`ConfigParser` object.

This method processes arbitrary parsers, not only those read from a
single file, whereas :meth:`.ManimConfig.digest_file` can only process one
file at a time.

Parameters
-----
parser
    An object reflecting the contents of one or many ``.cfg`` files. In
    particular, it may reflect the contents of multiple files that have
    been parsed in a cascading fashion.

Returns
-----
self : :class:`ManimConfig`
    This object, after processing the contents of ``parser``.

See Also
-----
:func:`make_config_parser`, :meth:`.ManimConfig.digest_file`,
:meth:`.ManimConfig.digest_args`,

Notes
-----

```

```
If there are multiple ``.cfg`` files to process, it is always more
efficient to parse them into a single :class:`ConfigParser` object
first, and then call this function once (instead of calling
:meth:`~.ManimConfig.digest_file` multiple times).
```

Examples

```
To digest the config options set in two files, first create a
ConfigParser and parse both files and then digest the parser:
```

```
.. code-block:: python

    parser = configparser.ConfigParser()
    parser.read([file1, file2])
    config = ManimConfig().digest_parser(parser)
```

In fact, the global ``config`` object is initialized like so:

```
.. code-block:: python

    parser = make_config_parser()
    config = ManimConfig().digest_parser(parser)

"""

self._parser = parser

# boolean keys
for key in [
    "notify_outdated_version",
    "write_to_movie",
    "save_last_frame",
    "write_all",
    "save_pngs",
    "save_as_gif",
    "save_sections",
    "preview",
    "show_in_file_browser",
    "log_to_file",
    "disable_caching",
    "disable_caching_warning",
    "flush_cache",
    "custom_folders",
    "enable_gui",
    "fullscreen",
    "use_projection_fill_shaders",
    "use_projection_stroke_shaders",
    "enable_wireframe",
    "force_window",
]:
    setattr(self, key, parser["CLI"].getboolean(key, fallback=False))

# int keys
for key in [
    "from_animation_number",
    "upto_animation_number",
    "max_files_cached",
    # the next two must be set BEFORE digesting frame_width and frame_height
    "pixel_height",
    "pixel_width",
    "window_monitor",
    "zero_pad",
]:
    setattr(self, key, parser["CLI"].getint(key))

# str keys
for key in [
    "assets_dir",
    "verbosity",
    "media_dir",
    "log_dir",
    "video_dir",
    "sections_dir",
    "images_dir",
    "text_dir",
    "tex_dir",
    "partial_movie_dir",
    "input_file",
    "output_file",
    "movie_file_extension",
    "background_color",
    "renderer",
    "window_position",
]:
    setattr(self, key, parser["CLI"].get(key, fallback="", raw=True))

# float keys
for key in [
    "background_opacity",
    "frame_rate",
    # the next two are floats but have their own logic, applied later
    # "frame_width",
    # "frame_height",
]:
    setattr(self, key, parser["CLI"].getfloat(key))

# tuple keys
gui_location = tuple(
    map(int, re.split(r";|\-]", parser["CLI"]["gui_location"])),
)
setattr(self, "gui_location", gui_location)

window_size = parser["CLI"][
    "window_size"
] # if not "default", get a tuple of the position
if window_size != "default":
    window_size = tuple(map(int, re.split(r";|\-]", window_size)))
setattr(self, "window_size", window_size)

# plugins
self.plugins = parser["CLI"].get("plugins", fallback="", raw=True).split(",")
# the next two must be set AFTER digesting pixel_width and pixel_height
self["frame_height"] = parser["CLI"].getfloat("frame_height", 8.0)
width = parser["CLI"].getfloat("frame_width", None)
if width is None:
    self["frame_width"] = self["frame_height"] * self["aspect_ratio"]
else:
    self["frame_width"] = width

# other logic
val = parser["CLI"].get("tex_template_file")
if val:
    self.tex_template_file = val

val = parser["CLI"].get("progress_bar")
if val:
    setattr(self, "progress_bar", val)

val = parser["ffmpeg"].get("loglevel")
if val:
    self.ffmpeg_loglevel = val

# TODO: Fix the mess above and below
val = parser["ffmpeg"].get("ffmpeg_executable")
setattr(self, "ffmpeg_executable", val)
```

```

try:
    val = parser["jupyter"].getboolean("media_embed")
except ValueError:
    val = None
setattr(self, "media_embed", val)

val = parser["jupyter"].get("media_width")
if val:
    setattr(self, "media_width", val)

val = parser["CLI"].get("quality", fallback="", raw=True)
if val:
    self.quality = _determine_quality(val)

return self

def digest_args(self, args: argparse.Namespace) -> ManimConfig:
    """Process the config options present in CLI arguments.

    Parameters
    -----
    args
        An object returned by :func:`.main_utils.parse_args()`.

    Returns
    -----
    self : :class:`ManimConfig`
        This object, after processing the contents of ``parser``.

    See Also
    -----
    :func:`.main_utils.parse_args()`, :meth:`~.ManimConfig.digest_parser`,
    :meth:`~.ManimConfig.digest_file`

    Notes
    -----
    If ``args.config_file`` is a non-empty string, ``ManimConfig`` tries to digest the
    contents of said file with :meth:`~.ManimConfig.digest_file` before
    digesting any other CLI arguments.

    """
    # if the input file is a config file, parse it properly
    if args.file.suffix == ".cfg":
        args.config_file = args.file

    # if args.file is ``, the animation code has to be taken from STDIN, so the
    # input file path shouldn't be absolute, since that file won't be read.
    if str(args.file) == "":
        self.input_file = args.file

    # if a config file has been passed, digest it first so that other CLI
    # flags supersede it
    if args.config_file:
        self.digest_file(args.config_file)

    # read input_file from the args if it wasn't set by the config file
    if not self.input_file:
        self.input_file = Path(args.file).absolute()

    self.scene_names = args.scene_names if args.scene_names is not None else []
    self.output_file = args.output_file

    for key in [
        "notify_outdated_version",
        "preview",
        "show_in_file_browser",
        "write_to_movie",
        "save_last_frame",
        "save_pngs",
        "save_as_gif",
        "save_sections",
        "write_all",
        "disable_caching",
        "format",
        "flush_cache",
        "progress_bar",
        "transparent",
        "scene_names",
        "verbosity",
        "renderer",
        "background_color",
        "enable_gui",
        "fullscreen",
        "use_projection_fill_shaders",
        "use_projection_stroke_shaders",
        "zero_pad",
        "enable_wireframe",
        "force_window",
        "dry_run",
    ]:
        if hasattr(args, key):
            attr = getattr(args, key)
            # if attr is None, then no argument was passed and we should
            # not change the current config
            if attr is not None:
                self[key] = attr

    for key in [
        "media_dir",  # always set this one first
        "log_dir",
        "log_to_file",  # always set this one last
    ]:
        if hasattr(args, key):
            attr = getattr(args, key)
            # if attr is None, then no argument was passed and we should
            # not change the current config
            if attr is not None:
                self[key] = attr

    if self["save_last_frame"]:
        self["write_to_movie"] = False

    # Handle the -n flag.
    nflag = args.from_animation_number
    if nflag:
        self.from_animation_number = nflag[0]
        try:
            self.upto_animation_number = nflag[1]
        except Exception:
            logging.getLogger("manim").info(
                f"No end scene number specified in -n option. Rendering from {nflag[0]} onwards...",
            )

    # Handle the quality flags
    self.quality = _determine_quality(getattr(args, "quality", None))

    # Handle the -r flag.
    rflag = args.resolution
    if rflag:
        self.pixel_width = int(rflag[0])
        self.pixel_height = int(rflag[1]))

    fps = args.frame_rate
    if fps:
        self.frame_rate = float(fps)

```

```

# Handle --custom_folders
if args.custom_folders:
    for opt in [
        "media_dir",
        "video_dir",
        "sections_dir",
        "images_dir",
        "text_dir",
        "tex_dir",
        "log_dir",
        "partial_movie_dir",
    ]:
        self[opt] = self._parser["custom_folders"].get(opt, raw=True)
    # --media_dir overrides the default.cfg file
    if hasattr(args, "media_dir") and args.media_dir:
        self.media_dir = args.media_dir

# Handle --tex_template
if args.tex_template:
    self.tex_template = TexTemplateFromFile(tex_filename=args.tex_template)

if (
    self.renderer == RendererType.OPENGL
    and getattr(args, "write_to_movie") is None
):
    # --write_to_movie was not passed on the command line, so don't generate video.
    self["write_to_movie"] = False

# Handle --gui_location flag.
if getattr(args, "gui_location") is not None:
    self.gui_location = args.gui_location

return self

def digest_file(self, filename: str | os.PathLike) -> ManimConfig:
    """Process the config options present in a ``.cfg`` file.

    This method processes a single ``.cfg`` file, whereas
    :meth:`~ManimConfig.digest_parser` can process arbitrary parsers, built
    perhaps from multiple ``.cfg`` files.

    Parameters
    -----
    filename
        Path to the ``.cfg`` file.

    Returns
    -----
    self : :class:`ManimConfig`
        This object, after processing the contents of ``filename``.

    See Also
    -----
    :meth:`~ManimConfig.digest_file`, :meth:`~ManimConfig.digest_args`,
    :func:`make_config_parser`

    Notes
    -----
    If there are multiple ``.cfg`` files to process, it is always more
    efficient to parse them into a single :class:`ConfigParser` object
    first and digesting them with one call to
    :meth:`~ManimConfig.digest_parser`, instead of calling this method
    multiple times.

    """
    if not Path(filename).is_file():
        raise FileNotFoundError(
            errno.ENOENT,
            "Error: --config_file could not find a valid config file.",
            str(filename),
        )

    return self.digest_parser(make_config_parser(filename))

# config options are properties
preview = property(
    lambda self: self._d["preview"] or self._d["enable_gui"],
    lambda self, val: self._set_boolean("preview", val),
    doc="Whether to play the rendered movie (-p).",
)

show_in_file_browser = property(
    lambda self: self._d["show_in_file_browser"],
    lambda self, val: self._set_boolean("show_in_file_browser", val),
    doc="Whether to show the output file in the file browser (-f).",
)

progress_bar = property(
    lambda self: self._d["progress_bar"],
    lambda self, val: self._set_from_list(
        "progress_bar",
        val,
        ["none", "display", "leave"],
    ),
    doc="Whether to show progress bars while rendering animations.",
)

log_to_file = property(
    lambda self: self._d["log_to_file"],
    lambda self, val: self._set_boolean("log_to_file", val),
    doc="Whether to save logs to a file.",
)

notify_outdated_version = property(
    lambda self: self._d["notify_outdated_version"],
    lambda self, val: self._set_boolean("notify_outdated_version", val),
    doc="Whether to notify if there is a version update available.",
)

write_to_movie = property(
    lambda self: self._d["write_to_movie"],
    lambda self, val: self._set_boolean("write_to_movie", val),
    doc="Whether to render the scene to a movie file (-w).",
)

save_last_frame = property(
    lambda self: self._d["save_last_frame"],
    lambda self, val: self._set_boolean("save_last_frame", val),
    doc="Whether to save the last frame of the scene as an image file (-s).",
)

write_all = property(
    lambda self: self._d["write_all"],
    lambda self, val: self._set_boolean("write_all", val),
    doc="Whether to render all scenes in the input file (-a).",
)

save_pngs = property(
    lambda self: self._d["save_pngs"],
    lambda self, val: self._set_boolean("save_pngs", val),
    doc="Whether to save all frames in the scene as images files (-g).",
)

save_as_gif = property(

```

```

        lambda self: self._d["save_as_gif"],
        lambda self, val: self._set_boolean("save as gif", val),
        doc="Whether to save the rendered scene in .gif format (-i).",
    )

    save_sections = property(
        lambda self: self._d["save_sections"],
        lambda self, val: self._set_boolean("save_sections", val),
        doc="Whether to save single videos for each section in addition to the movie file.",
    )

    enable_wireframe = property(
        lambda self: self._d["enable_wireframe"],
        lambda self, val: self._set_boolean("enable_wireframe", val),
        doc="Enable wireframe debugging mode in opengl.",
    )

    force_window = property(
        lambda self: self._d["force_window"],
        lambda self, val: self._set_boolean("force_window", val),
        doc="Set to force window when using the opengl renderer",
    )

@property
def verbosity(self):
    """Logger verbosity: "DEBUG", "INFO", "WARNING", "ERROR", or "CRITICAL" (-v)."""
    return self._d["verbosity"]

@verbosity.setter
def verbosity(self, val: str) -> None:
    """Verbosity level of the logger."""
    self._set_from_list(
        "verbosity",
        val,
        ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"],
    )
    logging.getLogger("manim").setLevel(val)

@property
def format(self):
    """File format: "png", "gif", "mp4", "webm" or "mov"."""
    return self._d["format"]

@format.setter
def format(self, val: str) -> None:
    """File format the renderer will output."""
    self._set_from_list(
        "format",
        val,
        [None, "png", "gif", "mp4", "mov", "webm"],
    )
    if self.format == "webm":
        logging.getLogger("manim").warning(
            "Output format set as webm, this can be slower than other formats",
        )

ffmpeg_loglevel = property(
    lambda self: self._d["ffmpeg_loglevel"],
    lambda self, val: self._set_from_list(
        "ffmpeg_loglevel",
        val,
        ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"],
    ),
    doc="Verbosity level of ffmpeg (no flag).",
)

ffmpeg_executable = property(
    lambda self: self._d["ffmpeg_executable"],
    lambda self, val: self._set_str("ffmpeg_executable", val),
    doc="Manually specify the path to the ffmpeg executable",
)

media_embed = property(
    lambda self: self._d["media_embed"],
    lambda self, val: self._setitem_("media_embed", val),
    doc="Embed videos in Jupyter notebook",
)

media_width = property(
    lambda self: self._d["media_width"],
    lambda self, val: self._setitem_("media_width", val),
    doc="Media width in Jupyter notebook",
)

pixel_width = property(
    lambda self: self._d["pixel_width"],
    lambda self, val: self._set_pos_number("pixel_width", val, False),
    doc="Frame width in pixels (--resolution, -r).",
)

pixel_height = property(
    lambda self: self._d["pixel_height"],
    lambda self, val: self._set_pos_number("pixel_height", val, False),
    doc="Frame height in pixels (--resolution, -r).",
)

aspect_ratio = property(
    lambda self: self._d["pixel_width"] / self._d["pixel_height"],
    doc="Aspect ratio (width / height) in pixels (--resolution, -r).",
)

frame_height = property(
    lambda self: self._d["frame_height"],
    lambda self, val: self._setitem_("frame_height", val),
    doc="Frame height in logical units (no flag).",
)

frame_width = property(
    lambda self: self._d["frame_width"],
    lambda self, val: self._setitem_("frame_width", val),
    doc="Frame width in logical units (no flag).",
)

frame_y_radius = property(
    lambda self: self._d["frame_height"] / 2,
    lambda self, val: (
        self._d._setitem_("frame_y_radius", val)
        or self._d._setitem_("frame_height", 2 * val)
    ),
    doc="Half the frame height (no flag).",
)

frame_x_radius = property(
    lambda self: self._d["frame_width"] / 2,
    lambda self, val: (
        self._d._setitem_("frame_x_radius", val)
        or self._d._setitem_("frame_width", 2 * val)
    ),
    doc="Half the frame width (no flag).",
)

top = property(
    lambda self: self.frame_y_radius * constants.UP,
    doc="Coordinate at the center top of the frame.",
)

```

```

bottom = property(
    lambda self: self.frame_y_radius * constants.DOWN,
    doc="Coordinate at the center bottom of the frame.",
)
left_side = property(
    lambda self: self.frame_x_radius * constants.LEFT,
    doc="Coordinate at the middle left of the frame.",
)
right_side = property(
    lambda self: self.frame_x_radius * constants.RIGHT,
    doc="Coordinate at the middle right of the frame.",
)
frame_rate = property(
    lambda self: self.d["frame_rate"],
    lambda self, val: self.d.__setitem__("frame_rate", val),
    doc="Frame rate in frames per second.",
)
background_color = property(
    lambda self: self.d["background_color"],
    lambda self, val: self.d.__setitem__("background_color", colour.Color(val)),
    doc="Background color of the scene (-c).",
)
from_animation_number = property(
    lambda self: self.d["from_animation_number"],
    lambda self, val: self.d.__setitem__("from_animation_number", val),
    doc="Start rendering animations at this number (-n).",
)
upto_animation_number = property(
    lambda self: self.d["upto_animation_number"],
    lambda self, val: self.d.__setitem__("upto_animation_number", val, True),
    doc="Stop rendering animations at this number. Use -1 to avoid skipping (-n).",
)
max_files_cached = property(
    lambda self: self.d["max_files_cached"],
    lambda self, val: self.d.__set_pos_number("max_files_cached", val, True),
    doc="Maximum number of files cached. Use -1 for infinity (no flag).",
)
window_monitor = property(
    lambda self: self.d["window_monitor"],
    lambda self, val: self.d.__set_pos_number("window_monitor", val, True),
    doc="The monitor on which the scene will be rendered",
)
flush_cache = property(
    lambda self: self.d["flush_cache"],
    lambda self, val: self.d.__set_boolean("flush_cache", val),
    doc="Whether to delete all the cached partial movie files.",
)
disable_caching = property(
    lambda self: self.d["disable_caching"],
    lambda self, val: self.d.__set_boolean("disable_caching", val),
    doc="Whether to use scene caching.",
)
disable_caching_warning = property(
    lambda self: self.d["disable_caching_warning"],
    lambda self, val: self.d.__set_boolean("disable_caching_warning", val),
    doc="Whether a warning is raised if there are too much subobjects to hash.",
)
movie_file_extension = property(
    lambda self: self.d["movie_file_extension"],
    lambda self, val: self.d.__set_from_list(
        "movie_file_extension",
        val,
        [".mp4", ".mov", ".webm"],
    ),
    doc="Either .mp4, .webm or .mov.",
)
background_opacity = property(
    lambda self: self.d["background_opacity"],
    lambda self, val: self.d.__set_between("background_opacity", val, 0, 1),
    doc="A number between 0.0 (fully transparent) and 1.0 (fully opaque).",
)
frame_size = property(
    lambda self: (self.d["pixel_width"], self.d["pixel_height"]),
    lambda self, tup: (
        self.d.__setitem__("pixel_width", tup[0])
        or self.d.__setitem__("pixel_height", tup[1])
    ),
    doc="Tuple with (pixel width, pixel height) (no flag).",
)
@property
def quality(self):
    """Video quality (-q)"""
    keys = ["pixel_width", "pixel_height", "frame_rate"]
    q = {k: self[k] for k in keys}
    for qual in constants.QUALITIES:
        if all(q[k] == constants.QUALITIES[qual][k] for k in keys):
            return qual
    return None

@quality.setter
def quality(self, qual: str) -> None:
    if qual is None:
        return
    if qual not in constants.QUALITIES:
        raise KeyError(f"quality must be one of {list(constants.QUALITIES.keys())}")
    q = constants.QUALITIES[qual]
    self.frame_size = q["pixel_width"], q["pixel_height"]
    self.frame_rate = q["frame_rate"]

@property
def transparent(self):
    """Whether the background opacity is 0.0 (-t)."""
    return self.d["background_opacity"] == 0.0

@transparent.setter
def transparent(self, val: bool) -> None:
    self.d["background_opacity"] = float(not val)
    self.resolve_movie_file_extension(val)

@property
def dry_run(self):
    """Whether dry run is enabled."""
    return self.d["dry_run"]

@dry_run.setter
def dry_run(self, val: bool) -> None:
    self.d["dry_run"] = val
    if val:
        self.write_to_movie = False
        self.write_all = False

```

```

        self.save_last_frame = False
        self.format = None

@property
def renderer(self):
    """The currently active renderer.

Populated with one of the available renderers in :class:`.RendererType`.

Tests::

    >>> test_config = ManimConfig()
    >>> test_config.renderer is None # a new ManimConfig is unpopulated
    True
    >>> test_config.renderer = 'opengl'
    >>> test_config.renderer
    <RendererType.OPENGL: 'opengl'>
    >>> test_config.renderer = 42
    Traceback (most recent call last):
    ...
    ValueError: 42 is not a valid RendererType

Check that capitalization of renderer types is irrelevant::

    >>> test_config.renderer = 'OpenGL'
    >>> test_config.renderer = 'cAirO'
    ...
    return self._d["renderer"]

@renderer.setter
def renderer(self, val: str | RendererType) -> None:
    """The setter of the renderer property.

Takes care of switching inheritance bases using the
:class:`.ConvertToOpenGL` metaclass.

    """
    if isinstance(val, str):
        val = val.lower()
    renderer = RendererType(val)
    try:
        from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
        from manim.mobject.opengl_mobject import OpenGLMobject
        from manim.mobject.opengl_vectorized_mobject import OpenGLVMOBJECT

        from ..mobject.mobject import Mobject
        from ..mobject.types.vectorized_mobject import VMobject

        for cls in ConvertToOpenGL.converted_classes:
            if renderer == RendererType.OPENGL:
                conversion_dict = {
                    Mobject: OpenGLMobject,
                    VMobject: OpenGLVMOBJECT,
                }
            else:
                conversion_dict = {
                    OpenGLMobject: Mobject,
                    OpenGLVMOBJECT: VMobject,
                }
            cls.__bases__ = tuple(
                conversion_dict.get(base, base) for base in cls.__bases__
            )
    except ImportError:
        # The renderer is set during the initial import of the
        # library for the first time. The imports above cause an
        # ImportError due to circular imports. However, the
        # metaclass sets stuff up correctly in this case, so we
        # can just do nothing.
        pass

    self._set_from_enum("renderer", renderer, RendererType)

media_dir = property(
    lambda self: self._d["media_dir"],
    lambda self, val: self._set_dir("media_dir", val),
    doc="Main output directory. See :meth:`ManimConfig.get_dir`.",
)

window_position = property(
    lambda self: self._d["window_position"],
    lambda self, val: self._setitem_("_window_position", val),
    doc="Set the position of preview window. You can use directions, e.g. UL/DR/ORIGIN/LEFT...or the position(pixel) of the upper left corner of the window, e.g. '960,540'.",
)

window_size = property(
    lambda self: self._d["window_size"],
    lambda self, val: self._setitem_("_window_size", val),
    doc="The size of the opengl window. 'default' to automatically scale the window based on the display monitor.",
)

def resolve_movie_file_extension(self, is_transparent):
    if is_transparent:
        self.movie_file_extension = ".webm" if self.format == "webm" else ".mov"
    elif self.format == "webm":
        self.movie_file_extension = ".webm"
    elif self.format == "mov":
        self.movie_file_extension = ".mov"
    else:
        self.movie_file_extension = ".mp4"

enable_gui = property(
    lambda self: self._d["enable_gui"],
    lambda self, val: self._set_boolean("enable_gui", val),
    doc="Enable GUI interaction.",
)

gui_location = property(
    lambda self: self._d["gui_location"],
    lambda self, val: self._set_tuple("gui_location", val),
    doc="Enable GUI interaction.",
)

fullscreen = property(
    lambda self: self._d["fullscreen"],
    lambda self, val: self._set_boolean("fullscreen", val),
    doc="Expand the window to its maximum possible size.",
)

use_projection_fill_shaders = property(
    lambda self: self._d["use_projection_fill_shaders"],
    lambda self, val: self._set_boolean("use_projection_fill_shaders", val),
    doc="Use shaders for OpenGLMobject fill which are compatible with transformation matrices.",
)

use_projection_stroke_shaders = property(
    lambda self: self._d["use_projection_stroke_shaders"],
    lambda self, val: self._set_boolean("use_projection_stroke_shaders", val),
    doc="Use shaders for OpenGLVMOBJECT stroke which are compatible with transformation matrices.",
)

zero_pad = property(
    lambda self: self._d["zero_pad"],
    lambda self, val: self._set_int_between("zero_pad", val, 0, 9),
    doc="PNG zero padding. A number between 0 (no zero padding) and 9 (9 columns minimum).",
)

```

```

def get_dir(self, key: str, **kwargs: str) -> Path:
    """Resolve a config option that stores a directory.

    Config options that store directories may depend on one another. This
    method is used to provide the actual directory to the end user.

    Parameters
    -----
    key
        The config option to be resolved. Must be an option ending in
        ``'_dir```, for example ``'media_dir'`` or ``'video_dir'``.

    kwargs
        Any strings to be used when resolving the directory.

    Returns
    -----
    :class:`pathlib.Path`
        Path to the requested directory. If the path resolves to the empty
        string, return ``None`` instead.

    Raises
    -----
    :class:`KeyError`
        When ``key`` is not a config option that stores a directory and
        thus :meth:`~ManimConfig.get_dir` is not appropriate; or when
        ``key`` is appropriate but there is not enough information to
        resolve the directory.

    Notes
    -----
    Standard :meth:`str.format` syntax is used to resolve the paths so the
    paths may contain arbitrary placeholders using f-string notation.
    However, these will require ``kwargs`` to contain the required values.

    Examples
    -----
    The value of ``config.tex_dir`` is ``'{media_dir}/Tex`` by default,
    i.e. it is a subfolder of wherever ``config.media_dir`` is located. In
    order to get the *actual* directory, use :meth:`~ManimConfig.get_dir`.

    .. code-block:: pycon

        >>> from manim import config as globalconfig
        >>> config = globalconfig.copy()
        >>> config.tex_dir
        '{media_dir}/Tex'
        >>> config.media_dir
        './media'
        >>> config.get_dir("tex_dir").as_posix()
        'media/Tex'

    Resolving directories is done in a lazy way, at the last possible
    moment, to reflect any changes in other config options:

    .. code-block:: pycon

        >>> config.media_dir = "my_media_dir"
        >>> config.get_dir("tex_dir").as_posix()
        'my_media_dir/Tex'

    Some directories depend on information that is not available to
    :class:`ManimConfig`. For example, the default value of ``video_dir``
    includes the name of the input file and the video quality
    (e.g. 480p15). This information has to be supplied via ``kwargs``:

    .. code-block:: pycon

        >>> config.video_dir
        '{media_dir}/videos/{module_name}/{quality}'
        >>> config.get_dir("video_dir")
        Traceback (most recent call last):
        KeyError: 'video_dir {media_dir}/videos/{module_name}/{quality} requires the following keyword arguments: module_name'
        >>> config.get_dir("video_dir", module_name="myfile").as_posix()
        'my_media_dir/videos/myfile/1080p60'

    Note the quality does not need to be passed as keyword argument since
    :class:`ManimConfig` does store information about quality.

    Directories may be recursively defined. For example, the config option
    ``partial_movie_dir`` depends on ``video_dir``, which in turn depends
    on ``media_dir``:

    .. code-block:: pycon

        >>> config.partial_movie_dir
        '{video_dir}/partial_movie_files/{scene_name}'
        >>> config.get_dir("partial_movie_dir")
        Traceback (most recent call last):
        KeyError: 'partial_movie_dir {video_dir}/partial_movie_files/{scene_name} requires the following keyword arguments: scene_name'
        >>> config.get_dir(
        ...     "partial_movie_dir", module_name="myfile", scene_name="myscene"
        ... ).as_posix()
        'my_media_dir/videos/myfile/1080p60/partial_movie_files/myscene'

    Standard f-string syntax is used. Arbitrary names can be used when
    defining directories, as long as the corresponding values are passed to
    :meth:`~ManimConfig.get_dir` via ``kwargs``.

    .. code-block:: pycon

        >>> config.media_dir = "{dir1}/{dir2}"
        >>> config.get_dir("media_dir")
        Traceback (most recent call last):
        KeyError: 'media_dir {dir1}/{dir2} requires the following keyword arguments: dir1'
        >>> config.get_dir("media_dir", dir1="foo", dir2="bar").as_posix()
        'foo/bar'
        >>> config.media_dir = "./media"
        >>> config.get_dir("media_dir").as_posix()
        'media'

    """
    dirs = [
        "assets_dir",
        "media_dir",
        "video_dir",
        "sections_dir",
        "images_dir",
        "text_dir",
        "tex_dir",
        "log_dir",
        "input_file",
        "output_file",
        "partial_movie_dir",
    ]
    if key not in dirs:
        raise KeyError(
            "must pass one of "
            "{media,video,images,text,tex,log}_dir "
            "or {input,output}_file",
        )
    dirs.remove(key) # a path cannot contain itself

```

```

all_args = {k: self._d[k] for k in dirs}
all_args.update(kwargs)
all_args["quality"] = f"(self.pixel_height)p{self.frame_rate:g}"

path = self._d[key]
while ":" in path:
    try:
        path = path.format(**all_args)
    except KeyError as exc:
        raise KeyError(
            f"(key) {self._d[key]} requires the following "
            + "keyword arguments: "
            + " ".join(exc.args),
        ) from exc
return Path(path) if path else None

def _set_dir(self, key: str, val: str | Path):
    if isinstance(val, Path):
        self._d.__setitem__(key, str(val))
    else:
        self._d.__setitem__(key, val)

assets_dir = property(
    lambda self: self._d["assets_dir"],
    lambda self, val: self._set_dir("assets_dir", val),
    doc="Directory to locate video assets (no flag).",
)

log_dir = property(
    lambda self: self._d["log_dir"],
    lambda self, val: self._set_dir("log_dir", val),
    doc="Directory to place logs. See :meth:`ManimConfig.get_dir`.",
)

video_dir = property(
    lambda self: self._d["video_dir"],
    lambda self, val: self._set_dir("video_dir", val),
    doc="Directory to place videos (no flag). See :meth:`ManimConfig.get_dir`.",
)

sections_dir = property(
    lambda self: self._d["sections_dir"],
    lambda self, val: self._set_dir("sections_dir", val),
    doc="Directory to place section videos (no flag). See :meth:`ManimConfig.get_dir`.",
)

images_dir = property(
    lambda self: self._d["images_dir"],
    lambda self, val: self._set_dir("images_dir", val),
    doc="Directory to place images (no flag). See :meth:`ManimConfig.get_dir`.",
)

text_dir = property(
    lambda self: self._d["text_dir"],
    lambda self, val: self._set_dir("text_dir", val),
    doc="Directory to place text (no flag). See :meth:`ManimConfig.get_dir`.",
)

tex_dir = property(
    lambda self: self._d["tex_dir"],
    lambda self, val: self._set_dir("tex_dir", val),
    doc="Directory to place tex (no flag). See :meth:`ManimConfig.get_dir`.",
)

partial_movie_dir = property(
    lambda self: self._d["partial_movie_dir"],
    lambda self, val: self._set_dir("partial_movie_dir", val),
    doc="Directory to place partial movie files (no flag). See :meth:`ManimConfig.get_dir`.",
)

custom_folders = property(
    lambda self: self._d["custom_folders"],
    lambda self, val: self._set_boolean("custom_folders", val),
    doc="Whether to use custom folder output.",
)

input_file = property(
    lambda self: self._d["input_file"],
    lambda self, val: self._set_dir("input_file", val),
    doc="Input file name.",
)

output_file = property(
    lambda self: self._d["output_file"],
    lambda self, val: self._set_dir("output_file", val),
    doc="Output file name (-o).",
)

scene_names = property(
    lambda self: self._d["scene_names"],
    lambda self, val: self._d.__setitem__("scene_names", val),
    doc="Scenes to play from file.",
)

@property
def tex_template(self):
    """Template used when rendering Tex. See :class:`.TexTemplate`."""
    if not hasattr(self, "_tex_template") or not self._tex_template:
        fn = self._d["tex_template_file"]
        if fn:
            self._tex_template = TexTemplateFromFile(tex_filename=fn)
        else:
            self._tex_template = TexTemplate()
    return self._tex_template

@tex_template.setter
def tex_template(self, val: TexTemplateFromFile | TexTemplate) -> None:
    if isinstance(val, (TexTemplateFromFile, TexTemplate)):
        self._tex_template = val

@property
def tex_template_file(self):
    """File to read Tex template from (no flag). See :class:`.TexTemplateFromFile`."""
    return self._d["tex_template_file"]

@tex_template_file.setter
def tex_template_file(self, val: str) -> None:
    if val:
        if not os.access(val, os.R_OK):
            logging.getLogger("manim").warning(
                f"Custom Tex template {val} not found or not readable.",
            )
    else:
        self._d["tex_template_file"] = Path(val)
    else:
        self._d["tex_template_file"] = val # actually set the falsy value

@property
def plugins(self):
    """List of plugins to enable."""
    return self._d["plugins"]

@plugins.setter
def plugins(self, value):

```

```

self._d["plugins"] = value

class ManimFrame(Mapping):
    _OPTS: set[str] = {
        "pixel_width",
        "pixel_height",
        "aspect_ratio",
        "frame_height",
        "frame_width",
        "frame_y_radius",
        "frame_x_radius",
        "top",
        "bottom",
        "left_side",
        "right_side",
    }
    _CONSTANTS: dict[str, np.ndarray] = {
        "UP": np.array((0.0, 1.0, 0.0)),
        "DOWN": np.array((0.0, -1.0, 0.0)),
        "RIGHT": np.array((1.0, 0.0, 0.0)),
        "LEFT": np.array((-1.0, 0.0, 0.0)),
        "IN": np.array((0.0, 0.0, -1.0)),
        "OUT": np.array((0.0, 0.0, 1.0)),
        "ORIGIN": np.array((0.0, 0.0, 0.0)),
        "X_AXIS": np.array((1.0, 0.0, 0.0)),
        "Y_AXIS": np.array((0.0, 1.0, 0.0)),
        "Z_AXIS": np.array((0.0, 0.0, 1.0)),
        "UL": np.array((-1.0, 1.0, 0.0)),
        "UR": np.array((1.0, 1.0, 0.0)),
        "DL": np.array((-1.0, -1.0, 0.0)),
        "DR": np.array((1.0, -1.0, 0.0)),
    }

    def __init__(self, c: ManimConfig) -> None:
        if not isinstance(c, ManimConfig):
            raise TypeError("Argument must be instance of 'ManimConfig'")
        # need to use __dict__ directly because setting attributes is not
        # allowed (see __setattr__)
        self.__dict__["_c"] = c

    # there are required by parent class Mapping to behave like a dict
    def __getitem__(self, key: str | int) -> Any:
        if key in self._OPTS:
            return self._c[key]
        elif key in self._CONSTANTS:
            return self._CONSTANTS[key]
        else:
            raise KeyError(key)

    def __iter__(self) -> Iterable:
        return iter(list(self._OPTS) + list(self._CONSTANTS))

    def __len__(self) -> int:
        return len(self._OPTS)

    # make this truly immutable
    def __setattr__(self, attr, val) -> None:
        raise TypeError("'ManimFrame' object does not support item assignment")

    def __setitem__(self, key, val) -> None:
        raise TypeError("'ManimFrame' object does not support item assignment")

    def __delitem__(self, key) -> None:
        raise TypeError("'ManimFrame' object does not support item deletion")

for opt in list(ManimFrame._OPTS) + list(ManimFrame._CONSTANTS):
    setattr(ManimFrame, opt, property(lambda self, o=opt: self[o]))

```

_test_class_makers.py

```
from __future__ import annotations

from typing import Callable

from manim.scene.scene import Scene
from manim.scene.scene_file_writer import SceneFileWriter

from ._frames_testers import _FramesTester


def _make_test_scene_class(
    base_scene: type[Scene],
    construct_test: Callable[[Scene], None],
    test_renderer,
) -> type[Scene]:
    class _TestedScene(base_scene):
        def __init__(self, *args, **kwargs):
            super().__init__(renderer=test_renderer, *args, **kwargs)

        def construct(self):
            construct_test(self)

            # Manim hack to render the very last frame (normally the last frame is not the very end of the animation)
            if self.animations is not None:
                self.update_to_time(self.get_run_time(self.animations))
                self.renderer.render(self, 1, self.moving_mobjects)

    return _TestedScene


def _make_test_renderer_class(from_renderer):
    # Just for inheritance.
    class _TestRenderer(from_renderer):
        pass

    return _TestRenderer


class DummySceneFileWriter(SceneFileWriter):
    """Delegate of SceneFileWriter used to test the frames."""

    def __init__(self, renderer, scene_name, **kwargs):
        super().__init__(renderer, scene_name, **kwargs)
        self.i = 0

    def init_output_directories(self, scene_name):
        pass

    def add_partial_movie_file(self, hash_animation):
        pass

    def begin_animation(self, allow_write=True):
        pass

    def end_animation(self, allow_write):
        pass

    def combine_to_movie(self):
        pass

    def combine_to_section_videos(self):
        pass

    def clean_cache(self):
        pass

    def write_frame(self, frame_or_renderer):
        self.i += 1


def _make_scene_file_writer_class(tester: _FramesTester) -> type[SceneFileWriter]:
    class TestSceneFileWriter(DummySceneFileWriter):
        def write_frame(self, frame_or_renderer):
            tester.check_frame(self.i, frame_or_renderer)
            super().write_frame(frame_or_renderer)

    return TestSceneFileWriter
```

creation.py

```
r"""Animate the display or removal of a mobject from a scene.

.. manim:: CreationModule
    :hide_source:

from manim import ManimBanner
class CreationModule(Scene):
    def construct(self):
        s1 = Square()
        s2 = Square()
        s3 = Square()
        s4 = Square()
        VGroup(s1, s2, s3, s4).set_x(0).arrange(buff=1.9).shift(UP)
        s5 = Square()
        s6 = Square()
        s7 = Square()
        VGroup(s5, s6, s7).set_x(0).arrange(buff=2.6).shift(2 * DOWN)
        t1 = Text("Write", font_size=24).next_to(s1, UP)
        t2 = Text("AddTextLetterByLetter", font_size=24).next_to(s2, UP)
        t3 = Text("Create", font_size=24).next_to(s3, UP)
        t4 = Text("Uncreate", font_size=24).next_to(s4, UP)
        t5 = Text("DrawBorderThenFill", font_size=24).next_to(s5, UP)
        t6 = Text("ShowIncreasingSubsets", font_size=22).next_to(s6, UP)
        t7 = Text("ShowSubmobjectsOneByOne", font_size=22).next_to(s7, UP)

        self.add(s1, s2, s3, s4, s5, s6, s7, t1, t2, t3, t4, t5, t6, t7)

        texts = [Text("manim", font_size=29), Text("manim", font_size=29)]
        texts[0].move_to(s1.get_center())
        texts[1].move_to(s2.get_center())
        self.add(*texts)

        objs = [ManimBanner().scale(0.25) for _ in range(5)]
        objs[0].move_to(s3.get_center())
        objs[1].move_to(s4.get_center())
        objs[2].move_to(s5.get_center())
        objs[3].move_to(s6.get_center())
        objs[4].move_to(s7.get_center())
        self.add(*objs)

        self.play(
            # text creation
            Write(texts[0]),
            AddTextLetterByLetter(texts[1]),
            # mobject creation
            Create(objs[0]),
            Uncreate(objs[1]),
            DrawBorderThenFill(objs[2]),
            ShowIncreasingSubsets(objs[3]),
            ShowSubmobjectsOneByOne(objs[4]),
            run_time=3,
        )
        self.wait()

"""

from __future__ import annotations

__all__ = [
    "Create",
    "Uncreate",
    "DrawBorderThenFill",
    "Write",
    "Unwrite",
    "ShowPartial",
    "ShowIncreasingSubsets",
    "SpiralIn",
    "AddTextLetterByLetter",
    "RemoveTextLetterByLetter",
    "ShowSubmobjectsOneByOne",
    "AddTextWordByWord",
]

import itertools as it
from typing import TYPE_CHECKING, Callable, Iterable, Sequence

import numpy as np
from colour import Color
```

```

if TYPE_CHECKING:
    from manim.mobject.text.text_mobject import Text

from manim.mobject.opengl.opengl_surface import OpenGLSurface
from manim.mobject.opengl.vectorized_mobject import OpenGLVMobject

from .. import config
from ..animation.animation import Animation
from ..animation.composition import Succession
from ..constants import TAU
from ..mobject.mobject import Group, Mobject
from ..mobject.types.vectorized_mobject import VMobject
from ..utils.bezier import integer_interpolate
from ..utils.rate_functions import double_smooth, linear, smooth


class ShowPartial(Animation):
    """Abstract class for Animations that show the VMobject partially.

    Raises
    -----
    :class:`TypeError`
        If ``mobject`` is not an instance of :class:`~.VMobject`.

    See Also
    -----
    :class:`Create`, :class:`~.ShowPassingFlash`

    """
    def __init__(self,
                 mobject: VMobject | OpenGLVMobject | OpenGLSurface | None,
                 **kwargs,
                 ):
        pointwise = getattr(mobject, "pointwiseBecomePartial", None)
        if not callable(pointwise):
            raise NotImplementedError("This animation is not defined for this Mobject.")
        super().__init__(mobject, **kwargs)

    def interpolate_submobject(self,
                               submobject: Mobject,
                               starting_submobject: Mobject,
                               alpha: float,
                               ) -> None:
        submobject.pointwiseBecomePartial(
            starting_submobject, *self._get_bounds(alpha)
        )

    def _get_bounds(self, alpha: float) -> None:
        raise NotImplementedError("Please use Create or ShowPassingFlash")



class Create(ShowPartial):
    """Incrementally show a VMobject.

    Parameters
    -----
    mobject
        The VMobject to animate.

    Raises
    -----
    :class:`TypeError`
        If ``mobject`` is not an instance of :class:`~.VMobject`.

    Examples
    -----
    .. manim:: CreateScene

        class CreateScene(Scene):
            def construct(self):
                self.play(Create(Square()))

    See Also
    -----
    :class:`~.ShowPassingFlash`

    """
    def __init__(self,
                 mobject: VMobject | OpenGLVMobject | OpenGLSurface,
                 lag_ratio: float = 1.0,
                 introducer: bool = True,
                 **kwargs,
                 )

```

```

) -> None:
    super().__init__(mobject, lag_ratio=lag_ratio, introducer=introducer, **kwargs)

def _get_bounds(self, alpha: float) -> tuple[int, float]:
    return (0, alpha)

class Uncreate(Create):
    """Like :class:`Create` but in reverse.

Examples
-----
.. manim:: ShowUncreate

    class ShowUncreate(Scene):
        def construct(self):
            self.play(Uncreate(Square()))

See Also
-----
:class:`Create`

"""

def __init__(
    self,
    mobject: VMobject | OpenGLVMOBJ,
    reverse_rate_function: bool = True,
    remover: bool = True,
    **kwargs,
) -> None:
    super().__init__(
        mobject,
        reverse_rate_function=reverse_rate_function,
        introducer=False,
        remover=remover,
        **kwargs,
    )

class DrawBorderThenFill(Animation):
    """Draw the border first and then show the fill.

Examples
-----
.. manim:: ShowDrawBorderThenFill

    class ShowDrawBorderThenFill(Scene):
        def construct(self):
            self.play(DrawBorderThenFill(Square(fill_opacity=1, fill_color=ORANGE)))
"""

def __init__(
    self,
    vmobject: VMobject | OpenGLVMOBJ,
    run_time: float = 2,
    rate_func: Callable[[float], float] = double_smooth,
    stroke_width: float = 2,
    stroke_color: str = None,
    draw_border_animation_config: dict = {}, # what does this dict accept?
    fill_animation_config: dict = {},
    introducer: bool = True,
    **kwargs,
) -> None:
    self._typecheck_input(vmobject)
    super().__init__(
        vmobject,
        run_time=run_time,
        introducer=introducer,
        rate_func=rate_func,
        **kwargs,
    )
    self.stroke_width = stroke_width
    self.stroke_color = stroke_color
    self.draw_border_animation_config = draw_border_animation_config
    self.fill_animation_config = fill_animation_config
    self.outline = self.get_outline()

def _typecheck_input(self, vmobject: VMobject | OpenGLVMOBJ) -> None:
    if not isinstance(vmobject, (VMobject, OpenGLVMOBJ)):
        raise TypeError("DrawBorderThenFill only works for vectorized Mobjects")

def begin(self) -> None:
    self.outline = self.get_outline()
    super().begin()

def get_outline(self) -> Mobject:

```

```

outline = self.mobject.copy()
outline.set_fill(opacity=0)
for sm in outline.family_members_with_points():
    sm.set_stroke(color=self.get_stroke_color(sm), width=self.stroke_width)
return outline

def get_stroke_color(self, vmobject: VMobject | OpenGLVMobject) -> Color:
    if self.stroke_color:
        return self.stroke_color
    elif vmobject.get_stroke_width() > 0:
        return vmobject.get_stroke_color()
    return vmobject.get_color()

def get_all_mobjects(self) -> Sequence[Mobject]:
    return [*super().get_all_mobjects(), self.outline]

def interpolate_submobject(
    self,
    submobject: Mobject,
    starting_submobject: Mobject,
    outline,
    alpha: float,
) -> None: # Fixme: not matching the parent class? What is outline doing here?
    index: int
    subalpha: int
    index, subalpha = integer_interpolate(0, 2, alpha)
    if index == 0:
        submobject.pointwise_become_partial(outline, 0, subalpha)
        submobject.match_style(outline)
    else:
        submobject.interpolate(outline, starting_submobject, subalpha)

class Write(DrawBorderThenFill):
    """Simulate hand-writing a :class:`Text` or hand-drawing a :class:`VMobject`."""

Examples
-----
.. manim:: ShowWrite

    class ShowWrite(Scene):
        def construct(self):
            self.play(Write(Text("Hello", font_size=144)))

.. manim:: ShowWriteReversed

    class ShowWriteReversed(Scene):
        def construct(self):
            self.play(Write(Text("Hello", font_size=144), reverse=True, remover=False))

Tests
-----
Check that creating empty :class:`Write` animations works::

    >>> from manim import Write, Text
    >>> Write(Text(''))
    Write(Text(''))
"""

def __init__(
    self,
    vmobject: VMobject | OpenGLVMobject,
    rate_func: Callable[[float], float] = linear,
    reverse: bool = False,
    **kwargs,
) -> None:
    run_time: float | None = kwargs.pop("run_time", None)
    lag_ratio: float | None = kwargs.pop("lag_ratio", None)
    run_time, lag_ratio = self._set_default_config_from_length(
        vmobject,
        run_time,
        lag_ratio,
    )
    self.reverse = reverse
    if "remover" not in kwargs:
        kwargs["remover"] = reverse
    super().__init__(
        vmobject,
        rate_func=rate_func,
        run_time=run_time,
        lag_ratio=lag_ratio,
        introducer=not reverse,
        **kwargs,
    )

def _set_default_config_from_length(

```

```

    self,
    vmobject: VMobject | OpenGLVMobject,
    run_time: float | None,
    lag_ratio: float | None,
) -> tuple[float, float]:
    length = len(vmobject.family_members_with_points())
    if run_time is None:
        if length < 15:
            run_time = 1
        else:
            run_time = 2
    if lag_ratio is None:
        lag_ratio = min(4.0 / max(1.0, length), 0.2)
    return run_time, lag_ratio

def reverse_subobjects(self) -> None:
    self.mobject.invert(recursive=True)

def begin(self) -> None:
    if self.reverse:
        self.reverse_subobjects()
    super().begin()

def finish(self) -> None:
    super().finish()
    if self.reverse:
        self.reverse_subobjects()

class Unwrite(Write):
    """Simulate erasing by hand a :class:`~.Text` or a :class:`~.VMobject`."""

    Parameters
    -----
    reverse
        Set True to have the animation start erasing from the last submobject first.

    Examples
    -----
    .. manim :: UnwriteReverseTrue

        class UnwriteReverseTrue(Scene):
            def construct(self):
                text = Tex("Alice and Bob").scale(3)
                self.add(text)
                self.play(Unwrite(text))

    .. manim:: UnwriteReverseFalse

        class UnwriteReverseFalse(Scene):
            def construct(self):
                text = Tex("Alice and Bob").scale(3)
                self.add(text)
                self.play(Unwrite(text, reverse=False))
    """

    def __init__(
        self,
        vmobject: VMobject,
        rate_func: Callable[[float], float] = linear,
        reverse: bool = True,
        **kwargs,
    ) -> None:
        run_time: float | None = kwargs.pop("run_time", None)
        lag_ratio: float | None = kwargs.pop("lag_ratio", None)
        run_time, lag_ratio = self._set_default_config_from_length(
            vmobject,
            run_time,
            lag_ratio,
        )
        super().__init__(
            vmobject,
            run_time=run_time,
            lag_ratio=lag_ratio,
            reverse_rate_function=True,
            reverse=reverse,
            **kwargs,
        )

    class SpiralIn(Animation):
        """Create the Mobject with sub-Mobjects flying in on spiral trajectories.

        Parameters
        -----
        shapes
    """

```

The Mobject on which to be operated.

scale_factor

The factor used for scaling the effect.

fade_in_fraction

Fractional duration of initial fade-in of sub-Mobjects as they fly inward.

Examples

.. manim :: SpiralInExample

```
class SpiralInExample(Scene):
```

```
    def construct(self):
```

```
        pi = MathTex(r"\pi").scale(7)
```

```
        pi.shift(2.25 * LEFT + 1.5 * UP)
```

```
        circle = Circle(color=GREEN_C, fill_opacity=1).shift(LEFT)
```

```
        square = Square(color=BLUE_D, fill_opacity=1).shift(UP)
```

```
        shapes = VGroup(pi, circle, square)
```

```
        self.play(SpiralIn(shapes))
```

```
"""
```

```
def __init__(
```

```
    self,
```

```
    shapes: Mobject,
```

```
    scale_factor: float = 8,
```

```
    fade_in_fraction=0.3,
```

```
    **kwargs,
```

```
) -> None:
```

```
    self.shapes = shapes
```

```
    self.scale_factor = scale_factor
```

```
    self.shape_center = shapes.get_center()
```

```
    self.fade_in_fraction = fade_in_fraction
```

```
    for shape in shapes:
```

```
        shape.final_position = shape.get_center()
```

```
        shape.initial_position = (
```

```
            shape.final_position
```

```
            + (shape.final_position - self.shape_center) * self.scale_factor
```

```
)
```

```
        shape.move_to(shape.initial_position)
```

```
        shape.save_state()
```

```
super().__init__(shapes, introducer=True, **kwargs)
```

```
def interpolate_mobject(self, alpha: float) -> None:
```

```
    alpha = self.rate_func(alpha)
```

```
    for shape in self.shapes:
```

```
        shape.restore()
```

```
        shape.save_state()
```

```
        opacity = shape.get_fill_opacity()
```

```
        new_opacity = min(opacity, alpha * opacity / self.fade_in_fraction)
```

```
        shape.shift((shape.final_position - shape.initial_position) * alpha)
```

```
        shape.rotate(TAU * alpha, about_point=self.shape_center)
```

```
        shape.rotate(-TAU * alpha, about_point=shape.get_center_of_mass())
```

```
        shape.set_opacity(new_opacity)
```

```
class ShowIncreasingSubsets(Animation):
```

```
    """Show one submobject at a time, leaving all previous ones displayed on screen.
```

Examples

.. manim:: ShowIncreasingSubsetsScene

```
class ShowIncreasingSubsetsScene(Scene):
```

```
    def construct(self):
```

```
        p = VGroup(Dot(), Square(), Triangle())
```

```
        self.add(p)
```

```
        self.play(ShowIncreasingSubsets(p))
```

```
        self.wait()
```

```
"""
```

```
def __init__(
```

```
    self,
```

```
    group: Mobject,
```

```
    suspend_mobject_updating: bool = False,
```

```
    int_func: Callable[[np.ndarray], np.ndarray] = np.floor,
```

```
    reverse_rate_function=False,
```

```
    **kwargs,
```

```
) -> None:
```

```
    self.all_submobs = list(group.submobjects)
```

```
    self.int_func = int_func
```

```
    for mob in self.all_submobs:
```

```
        mob.set_opacity(0)
```

```
super().__init__(
```

```
    group,
```

```

        suspend_mobject_updating=suspend_mobject_updating,
        reverse_rate_function=reverse_rate_function,
        **kwargs,
    )

def interpolate_mobject(self, alpha: float) -> None:
    n_submobs = len(self.all_submobs)
    value = (
        1 - self.rate_func(alpha)
        if self.reverse_rate_function
        else self.rate_func(alpha)
    )
    index = int(self.int_func(value * n_submobs))
    self.update_submobject_list(index)

def update_submobject_list(self, index: int) -> None:
    for mobj in self.all_submobs[:index]:
        mobj.set_opacity(1)
    for mobj in self.all_submobs[index:]:
        mobj.set_opacity(0)

class AddTextLetterByLetter(ShowIncreasingSubsets):
    """Show a :class:`~.Text` letter by letter on the scene.

    Parameters
    -----
    time_per_char
        Frequency of appearance of the letters.

    .. tip::
        This is currently only possible for class:`~.Text` and not for class:`~.MathTex`

    """

    def __init__(
        self,
        text: Text,
        suspend_mobject_updating: bool = False,
        int_func: Callable[[np.ndarray], np.ndarray] = np.ceil,
        rate_func: Callable[[float], float] = linear,
        time_per_char: float = 0.1,
        run_time: float | None = None,
        reverse_rate_function=False,
        introducer=True,
        **kwargs,
    ) -> None:
        self.time_per_char = time_per_char
        if run_time is None:
            # minimum time per character is 1/frame_rate, otherwise
            # the animation does not finish.
            run_time = np.max((1 / config.frame_rate, self.time_per_char)) * len(text)
        super().__init__(
            text,
            suspend_mobject_updating=suspend_mobject_updating,
            int_func=int_func,
            rate_func=rate_func,
            run_time=run_time,
            reverse_rate_function=reverse_rate_function,
            introducer=introducer,
            **kwargs,
        )

class RemoveTextLetterByLetter(AddTextLetterByLetter):
    """Remove a :class:`~.Text` letter by letter from the scene.

    Parameters
    -----
    time_per_char
        Frequency of appearance of the letters.

    .. tip::
        This is currently only possible for class:`~.Text` and not for class:`~.MathTex`

    """

    def __init__(
        self,
        text: Text,
        suspend_mobject_updating: bool = False,
        int_func: Callable[[np.ndarray], np.ndarray] = np.ceil,
        rate_func: Callable[[float], float] = linear,
        time_per_char: float = 0.1,
        run_time: float | None = None,
    
```

```

reverse_rate_function=True,
introducer=False,
remover=True,
**kwargs,
) -> None:
    super().__init__(
        text,
        suspend_mobject_updating=suspend_mobject_updating,
        int_func=int_func,
        rate_func=rate_func,
        time_per_char=time_per_char,
        run_time=run_time,
        reverse_rate_function=reverse_rate_function,
        introducer=introducer,
        remover=remover,
        **kwargs,
    )

class ShowSubmobjectsOneByOne(ShowIncreasingSubsets):
    """Show one submobject at a time, removing all previously displayed ones from screen."""

    def __init__(
        self,
        group: Iterable[Mobject],
        int_func: Callable[[np.ndarray], np.ndarray] = np.ceil,
        **kwargs,
    ) -> None:
        new_group = Group(*group)
        super().__init__(new_group, int_func=int_func, **kwargs)

    def update_submobject_list(self, index: int) -> None:
        current_submobjects = self.all_submobs[:index]
        for mob in current_submobjects[:-1]:
            mob.set_opacity(0)
        if len(current_submobjects) > 0:
            current_submobjects[-1].set_opacity(1)

# TODO, this is broken...
class AddTextWordByWord(Succession):
    """Show a :class:`~.Text` word by word on the scene. Note: currently broken."""

    def __init__(
        self,
        text_mobject: Text,
        run_time: float = None,
        time_per_char: float = 0.06,
        **kwargs,
    ) -> None:
        self.time_per_char = time_per_char
        tpc = self.time_per_char
        anims = it.chain(
            *(
                [
                    ShowIncreasingSubsets(word, run_time=tpc * len(word)),
                    Animation(word, run_time=0.005 * len(word) ** 1.5),
                ]
                for word in text_mobject
            )
        )
        super().__init__(*anims, **kwargs)

```

update.py

```
"""Animations that update mobjects."""

from __future__ import annotations

__all__ = ["UpdateFromFunc", "UpdateFromAlphaFunc", "MaintainPositionRelativeTo"]

import operator as op
import typing

from manim.animation.animation import Animation

if typing.TYPE_CHECKING:
    from manim.mobject.mobject import Mobject

class UpdateFromFunc(Animation):
    """
    update_function of the form func(mobject), presumably
    to be used when the state of one mobject is dependent
    on another simultaneously animated mobject
    """

    def __init__(
        self,
        mobject: Mobject,
        update_function: typing.Callable[[Mobject], typing.Any],
        suspend_mobject_updating: bool = False,
        **kwargs,
    ) -> None:
        self.update_function = update_function
        super().__init__(
            mobject, suspend_mobject_updating=suspend_mobject_updating, **kwargs
        )

    def interpolate_mobject(self, alpha: float) -> None:
        self.update_function(self.mobject)

class UpdateFromAlphaFunc(UpdateFromFunc):
    def interpolate_mobject(self, alpha: float) -> None:
        self.update_function(self.mobject, self.rate_func(alpha))

class MaintainPositionRelativeTo(Animation):
    def __init__(self, mobject: Mobject, tracked_mobject: Mobject, **kwargs) -> None:
        self.tracked_mobject = tracked_mobject
        self.diff = op.sub(
            mobject.get_center(),
            tracked_mobject.get_center(),
        )
        super().__init__(mobject, **kwargs)

    def interpolate_mobject(self, alpha: float) -> None:
        target = self.tracked_mobject.get_center()
        location = self.mobject.get_center()
        self.mobject.shift(target - location + self.diff)
```

hashing.py

```
"""Utilities for scene caching."""

from __future__ import annotations

import collections
import copy
import inspect
import json
import typing
import zlib
from time import perf_counter
from types import FunctionType, MappingProxyType, MethodType, ModuleType
from typing import Any

import numpy as np

from manim.animation.animation import Animation
from manim.camera.camera import Camera
from manim.mobject.mobject import Mobject

from .. import config, logger

if typing.TYPE_CHECKING:
    from manim.scene.scene import Scene

# Sometimes there are elements that are not suitable for hashing (too long or
# run-dependent). This is used to filter them out.
KEYS_TO_FILTER_OUT = {
    "original_id",
    "background",
    "pixel_array",
    "pixel_array_to_cairo_context",
}

class _Memoizer:
    """Implements the memoization logic to optimize the hashing procedure and prevent
    the circular references within iterable processed.

    Keeps a record of all the processed objects, and handle the logic to return a place
    holder instead of the original object if the object has already been processed
    by the hashing logic (i.e., recursively checked, converted to JSON, etc..).

    This class uses two signatures functions to keep a track of processed objects :
    hash or id. Whenever possible, hash is used to ensure a broader object
    content-equality detection.
    """

    _already_processed = set()

    # Can be changed to whatever string to help debugging the JSON generation.
    ALREADY_PROCESSED_PLACEHOLDER = "AP"
    THRESHOLD_WARNING = 170_000

    @classmethod
    def reset_already_processed(cls):
        cls._already_processed.clear()

    @classmethod
    def check_already_processed_decorator(cls: _Memoizer, is_method: bool = False):
        """Decorator to handle the arguments that goes through the decorated function.
        Returns _ALREADY_PROCESSED_PLACEHOLDER if the obj has been processed, or lets
        the decorated function call go ahead.

        Parameters
        -----
        is_method
            Whether the function passed is a method, by default False.
        """

        def layer(func):
            # NOTE : There is probably a better way to separate both case when func is
            # a method or a function.
            if is_method:
                return lambda self, obj: cls._handle_already_processed(
                    obj,
                    default_function=lambda obj: func(self, obj),
                )
            return lambda obj: cls._handle_already_processed(obj, default_function=func)

        return layer

    @classmethod
    def check_already_processed(cls, obj: Any) -> Any:
        """Checks if obj has been already processed. Returns itself if it has not been,
        or the value of _ALREADY_PROCESSED_PLACEHOLDER if it has.
        Marks the object as processed in the second case.

        Parameters
        -----
        obj
            The object to check.

        Returns
        -----
        Any
            Either the object itself or the placeholder.

        """
        # When the object is not memoized, we return the object itself.
        return cls._handle_already_processed(obj, lambda x: x)

    @classmethod
    def mark_as_processed(cls, obj: Any) -> None:
        """Marks an object as processed.

        Parameters
        -----
        obj
            The object to mark as processed.

        """
        cls._handle_already_processed(obj, lambda x: x)
        return cls._return(obj, id, lambda x: x, memoizing=False)

    @classmethod
    def _handle_already_processed(
        cls,
        obj,
        default_function: typing.Callable[[Any], Any],
    ):
        if isinstance(
            obj,
            (
                int,
                float,
                str,
                complex,
            ),
        ) and obj not in [None, cls.ALREADY_PROCESSED_PLACEHOLDER]:
            # It makes no sense (and it'd slower) to memoize objects of these primitive
            # types. Hence, we simply return the object.
            return obj
        if isinstance(obj, collections.abc.Hashable):
            try:
```

```

        return cls._return(obj, hash, default_function)
    except TypeError:
        # In case of an error with the hash (eg an object is marked as hashable
        # but contains a non hashable within it)
        # Fallback to use the built-in function id instead.
        pass
    return cls._return(obj, id, default_function)

@classmethod
def _return(
    cls,
    obj: typing.Any,
    obj_to_membership_sign: typing.Callable[[Any], int],
    default_func,
    memoizing=True,
) -> str | Any:
    obj_membership_sign = obj_to_membership_sign(obj)
    if obj_membership_sign in cls._already_processed:
        return cls.ALREADY_PROCESSED_PLACEHOLDER
    if memoizing:
        if (
            not config.disable_caching_warning
            and len(cls._already_processed) == cls.THRESHOLD_WARNING
        ):
            logger.warning(
                "It looks like the scene contains a lot of sub-mobjects. Caching "
                "is sometimes not suited to handle such large scenes, you might "
                "consider disabling caching with --disable_caching to potentially "
                "speed up the rendering process."
            )
            logger.warning(
                "You can disable this warning by setting disable_caching_warning "
                "to True in your config file."
            )
    cls._already_processed.add(obj_membership_sign)
    return default_func(obj)

class _CustomEncoder(json.JSONEncoder):
    def default(self, obj: Any):
        """
        This method is used to serialize objects to JSON format.

        If obj is a function, then it will return a dict with two keys : 'code', for
        the code source, and 'nonlocals' for all nonlocalsvalues. (including nonlocals
        functions, that will be serialized as this is recursive)
        if obj is a np.ndarray, it converts it into a list.
        if obj is an object with __dict__ attribute, it returns its __dict__.
        Else, will let the JSONEncoder do the stuff, and throw an error if the type is
        not suitable for JSONEncoder.

        Parameters
        -----
        obj
            Arbitrary object to convert

        Returns
        -----
        Any
            Python object that JSON encoder will recognize
        """
        if not (isinstance(obj, ModuleType) and isinstance(
            obj,
            (MethodType, FunctionType),
        )):
            cvars = inspect.getclosurevars(obj)
            cvardict = {**copy.deepcopy(cvars.globals), **copy.deepcopy(cvars.nonlocals)}
            for i in list(cvardict):
                # NOTE : All module types objects are removed, because otherwise it
                # throws ValueError: Circular reference detected if not. TODO
                if isinstance(cvardict[i], ModuleType):
                    del cvardict[i]
            try:
                code = inspect.getsource(obj)
            except (OSError, TypeError):
                # This happens when rendering videos included in the documentation
                # within doctests and should be replaced by a solution avoiding
                # hash collision (due to the same, empty, code strings) at some point.
                # See https://github.com/ManimCommunity/manim/pull/402.
                code = ""
            return self.cleaned_iterable({"code": code, "nonlocals": cvardict})
        elif isinstance(obj, np.ndarray):
            if obj.size > 1000:
                obj = np.resize(obj, (100, 100))
                return f"TRUNCATED ARRAY: {repr(obj)}"
            # We return the repr and not a list to avoid the JsonEncoder to iterate over it.
            return repr(obj)
        elif hasattr(obj, "__dict__"):
            temp = getattr(obj, "__dict__")
            # MappingProxy is scene-caching nightmare. It contains all of the object methods and attributes. We skip it as the mechanism will at some point process the object, but ins
            # Indeed, there is certainly no case where scene-caching will receive only a non instanced object, as this is never used in the library or encouraged to be used user-side
            if isinstance(temp, MappingProxyType):
                return "MappingProxy"
            return self.cleaned_iterable(temp)
        elif isinstance(obj, np.uint8):
            return int(obj)
        # Serialize it with only the type of the object. You can change this to whatever string when debugging the serialization process.
        return str(type(obj))

    def _cleaned_iterable(self, iterable: typing.Iterable[Any]):
        """
        Check for circular reference at each iterable that will go through the JSONEncoder, as well as key of the wrong format.

        If a key with a bad format is found (i.e not a int, string, or float), it gets replaced by its hash using the same process implemented here.
        If a circular reference is found within the iterable, it will be replaced by the string "already processed".
        """
        Parameters
        -----
        iterable
            The iterable to check.
        """
        def _key_to_hash(key):
            return zlib.crc32(json.dumps(key, cls=_CustomEncoder).encode())

        def _iter_check_list(lst):
            processed_list = [None] * len(lst)
            for i, el in enumerate(lst):
                el = _Memoizer.check_already_processed(el)
                if isinstance(el, (list, tuple)):
                    new_value = _iter_check_list(el)
                elif isinstance(el, dict):
                    new_value = _iter_check_dict(el)
                else:
                    new_value = el
                processed_list[i] = new_value
            return processed_list

        def _iter_check_dict(dct):
            processed_dict = {}
            for k, v in dct.items():
                v = _Memoizer.check_already_processed(v)
                if k in KEYS_TO_FILTER_OUT:
                    continue
                # We check if the k is of the right format (supporter by Json)
                if not isinstance(k, (str, int, float, bool)) and k is not None:
                    k_new = _key_to_hash(k)
                else:

```

```

        k_new = k
    if isinstance(v, dict):
        new_value = _iter_check_dict(v)
    elif isinstance(v, (list, tuple)):
        new_value = _iter_check_list(v)
    else:
        new_value = v
    processed_dict[k_new] = new_value
    return processed_dict

if isinstance(iterable, (list, tuple)):
    return _iter_check_list(iterable)
elif isinstance(iterable, dict):
    return _iter_check_dict(iterable)

def encode(self, obj: Any):
    """Overriding of :meth:`JSONEncoder.encode`, to make our own process.

    Parameters
    -----
    obj
        The object to encode in JSON.

    Returns
    -----
    :class:`str`
        The object encoder with the standard json process.
    """
    Memoizer.mark_as_processed(obj)
    if isinstance(obj, (dict, list, tuple)):
        return super().encode(self._cleaned_iterable(obj))
    return super().encode(obj)

def get_json(obj: dict):
    """Recursively serialize `object` to JSON using the :class:`CustomEncoder` class.

    Parameters
    -----
    obj
        The dict to flatten

    Returns
    -----
    :class:`str`
        The flattened object
    """
    return json.dumps(obj, cls=_CustomEncoder)

def get_hash_from_play_call(
    scene_object: Scene,
    camera_object: Camera,
    animations_list: typing.Iterable[Animation],
    current_mobjects_list: typing.Iterable[Mobject],
) -> str:
    """Take the list of animations and a list of mobjects and output their hashes. This is meant to be used for `scene.play` function.

    Parameters
    -----
    scene_object
        The scene object.

    camera_object
        The camera object used in the scene.

    animations_list
        The list of animations.

    current_mobjects_list
        The list of mobjects.

    Returns
    -----
    :class:`str`
        A string concatenation of the respective hashes of `camera_object`, `animations_list` and `current_mobjects_list`, separated by `_`.
    """
    logger.debug("Hashing ...")
    t_start = perf_counter()
    _Memoizer.mark_as_processed(scene_object)
    camera_json = get_json(camera_object)
    animations_list_json = [get_json(x) for x in sorted(animations_list, key=str)]
    current_mobjects_list_json = [get_json(x) for x in current_mobjects_list]
    hash_camera, hash_animations, hash_current_mobjects = (
        zlib.crc32(repr(json_val).encode())
        for json_val in [camera_json, animations_list_json, current_mobjects_list_json]
    )
    hash_complete = f"(hash_camera)_{(hash_animations)}_{(hash_current_mobjects)}"
    t_end = perf_counter()
    logger.debug("Hashing done in %(time)s s.", {"time": str(t_end - t_start)[:8]})
    # End of the hashing for the animation, reset all the memoize.
    _Memoizer.reset_already_processed()
    logger.debug("Hash generated : %(h)s", {"h": hash_complete})
    return hash_complete

```

group.py_1

```
from __future__ import annotations

import configparser
from pathlib import Path

import click
import cloup

from ... import console
from ...constants import CONTEXT_SETTINGS, EPILOG, QUALITIES
from ...utils.file_ops import (
    add_import_statement,
    copy_template_files,
    get_template_names,
    get_template_path,
)
CFG_DEFAULTS = {
    "frame_rate": 30,
    "background_color": "BLACK",
    "background_opacity": 1,
    "scene_names": "Default",
    "resolution": (854, 480),
}
def select_resolution():
    """Prompts input of type click.Choice from user. Presents options from QUALITIES constant.

    Returns
    ------
    :class:`tuple`
        Tuple containing height and width.
    """
    resolution_options = []
    for quality in QUALITIES.items():
        resolution_options.append(
            (quality[1]["pixel_height"], quality[1]["pixel_width"]),
        )
    resolution_options.pop()
    choice = click.prompt(
        "\nSelect resolution:\n",
        type=click.Choice([f"{i[0]}p" for i in resolution_options]),
        show_default=False,
        default="480p",
    )
    return [res for res in resolution_options if f"{res[0]}p" == choice][0]

def update_cfg(cfg_dict: dict, project_cfg_path: Path):
    """Updates the manim.cfg file after reading it from the project_cfg_path.

    Parameters
    ------
    cfg_dict
        values used to update manim.cfg found project_cfg_path.
    project_cfg_path
        Path of manim.cfg file.
    """
    config = configparser.ConfigParser()
    config.read(project_cfg_path)
    cli_config = config["CLI"]
    for key, value in cfg_dict.items():
        if key == "resolution":
            cli_config["pixel_height"] = str(value[0])
            cli_config["pixel_width"] = str(value[1])
        else:
            cli_config[key] = str(value)
    with project_cfg_path.open("w") as conf:
        config.write(conf)

@cloup.command(
    context_settings=CONTEXT_SETTINGS,
    epilog=EPILOG,
)
@cloup.argument("project_name", type=Path, required=False)
@cloup.option(
    "-d",
    "--default",
    "default_settings",
    is_flag=True,
    help="Default settings for project creation.",
    nargs=1,
)
def project(default_settings, **args):
    """Creates a new project.

    PROJECT_NAME is the name of the folder in which the new project will be initialized.
    """
    if args["project_name"]:
        project_name = args["project_name"]
    else:
        project_name = click.prompt("Project Name", type=Path)
```

```

# in the future when implementing a full template system. Choices are going to be saved in some sort of config file for templates
template_name = click.prompt(
    "Template",
    type=click.Choice(get_template_names(), False),
    default="Default",
)

if project_name.is_dir():
    console.print(
        f"\nFolder [red]{project_name}[/red] exists. Please type another name\n",
    )
else:
    project_name.mkdir()
    new_cfg = {}
    new_cfg_path = Path.resolve(project_name / "manim.cfg")

    if not default_settings:
        for key, value in CFG_DEFAULTS.items():
            if key == "scene_names":
                new_cfg[key] = template_name + "Template"
            elif key == "resolution":
                new_cfg[key] = select_resolution()
            else:
                new_cfg[key] = click.prompt(f"\n{key}", default=value)

        console.print("\n", new_cfg)
        if click.confirm("Do you want to continue?", default=True, abort=True):
            copy_template_files(project_name, template_name)
            update_cfg(new_cfg, new_cfg_path)
        else:
            copy_template_files(project_name, template_name)
            update_cfg(CFG_DEFAULTS, new_cfg_path)

@cloup.command(
    context_settings=CONTEXT_SETTINGS,
    no_args_is_help=True,
    epilog=EPILOG,
)
@cloup.argument("scene_name", type=str, required=True)
@cloup.argument("file_name", type=str, required=False)
def scene(**args):
    """Inserts a SCENE to an existing FILE or creates a new FILE.

    SCENE is the name of the scene that will be inserted.

    FILE is the name of file in which the SCENE will be inserted.
    """
    if not Path("main.py").exists():
        raise FileNotFoundError(f"Path('main.py') : Not a valid project directory.")

    template_name = click.prompt(
        "template",
        type=click.Choice(get_template_names(), False),
        default="Default",
    )
    scene = (get_template_path() / f"{template_name}.mtp").resolve().read_text()
    scene = scene.replace(template_name + "Template", args["scene_name"], 1)

    if args["file_name"]:
        file_name = Path(args["file_name"]) + ".py"

        if file_name.is_file():
            # file exists so we are going to append new scene to that file
            with file_name.open("a") as f:
                f.write("\n\n\n" + scene)
        else:
            # file does not exist so we create a new file, append the scene and prepend the import statement
            file_name.write_text("\n\n\n" + scene)

            add_import_statement(file_name)
    else:
        # file name is not provided so we assume it is main.py
        # if main.py does not exist we do not continue
        with Path("main.py").open("a") as f:
            f.write("\n\n\n" + scene)

@cloup.group(
    context_settings=CONTEXT_SETTINGS,
    invoke_without_command=True,
    no_args_is_help=True,
    epilog=EPILOG,
    help="Create a new project or insert a new scene.",
    deprecated=True,
)
@cloup.pass_context
def new(ctx):
    pass

new.add_command(project)
new.add_command(scene)

```

family_ops.py

```
from __future__ import annotations

import itertools as it

def extract_mobject_family_members(mobject_list, only_those_with_points=False):
    result = list(it.chain(*[mob.get_family() for mob in mobject_list]))
    if only_those_with_points:
        result = [mob for mob in result if mob.has_points()]
    return result

def restructure_list_to_exclude_certain_family_members(mobject_list, to_remove):
    """
    Removes anything in to_remove from mobject_list, but in the event that one of
    the items to be removed is a member of the family of an item in mobject_list,
    the other family members are added back into the list.

    This is useful in cases where a scene contains a group, e.g. Group(m1, m2, m3),
    but one of its subobjects is removed, e.g. scene.remove(m1), it's useful
    for the list of mobject_list to be edited to contain other subobjects, but not m1.
    """
    new_list = []
    to_remove = extract_mobject_family_members(to_remove)

    def add_safe_mobjects_from_list(list_to_examine, set_to_remove):
        for mob in list_to_examine:
            if mob in set_to_remove:
                continue
            intersect = set_to_remove.intersection(mob.get_family())
            if intersect:
                add_safe_mobjects_from_list(mob.submobjects, intersect)
            else:
                new_list.append(mob)

    add_safe_mobjects_from_list(mobject_list, set(to_remove))
    return new_list
```

commands.py

```
"""Manim's init subcommand.

Manim's init subcommand is accessed in the command-line interface via ``manim init``. Here you can specify options, subcommands, and subgroups for the init group.

"""
from __future__ import annotations

import configparser
from pathlib import Path

import click
import cloup

from ... import console
from ...constants import CONTEXT_SETTINGS, EPILOG, QUALITIES
from ...utils.file_ops import (
    add_import_statement,
    copy_template_files,
    get_template_names,
    get_template_path,
)
CFG_DEFAULTS = {
    "frame_rate": 30,
    "background_color": "BLACK",
    "background_opacity": 1,
    "scene_names": "Default",
    "resolution": (854, 480),
}
def select_resolution():
    """Prompts input of type click.Choice from user. Presents options from QUALITIES constant.

    Returns
    ------
        :class:`tuple`  

        Tuple containing height and width.
    """
    resolution_options = []
    for quality in QUALITIES.items():
        resolution_options.append(
            (quality[1]["pixel_height"], quality[1]["pixel_width"]),
        )
    resolution_options.pop()
    choice = click.prompt(
        "\nSelect resolution:\n",
        type=click.Choice([f"{i[0]}p" for i in resolution_options]),
        show_default=False,
        default="480p",
    )
    return [res for res in resolution_options if f"{res[0]}p" == choice][0]

def update_cfg(cfg_dict: dict, project_cfg_path: Path):
    """Updates the manim.cfg file after reading it from the project_cfg_path.

    Parameters
    ------
    cfg_dict  

        values used to update manim.cfg found project_cfg_path.
    project_cfg_path  

        Path of manim.cfg file.
    """
    config = configparser.ConfigParser()
    config.read(project_cfg_path)
    cli_config = config["CLI"]
    for key, value in cfg_dict.items():
        if key == "resolution":
            cli_config["pixel_height"] = str(value[0])
            cli_config["pixel_width"] = str(value[1])
        else:
            cli_config[key] = str(value)
    with project_cfg_path.open("w") as conf:
        config.write(conf)

    @cloup.command(
        context_settings=CONTEXT_SETTINGS,
        epilog=EPILOG,
    )
    @cloup.argument("project_name", type=Path, required=False)
    @cloup.option(
        "-d",
        "--default",
        "default_settings",
        is_flag=True,
        help="Default settings for project creation.",
        nargs=1,
    )
    def project(default_settings, **args):
        """Creates a new project.
```

```

PROJECT_NAME is the name of the folder in which the new project will be initialized.
"""
if args["project_name"]:
    project_name = args["project_name"]
else:
    project_name = click.prompt("Project Name", type=Path)

# in the future when implementing a full template system. Choices are going to be saved in some sort of config file for templates
template_name = click.prompt(
    "Template",
    type=click.Choice(get_template_names(), False),
    default="Default",
)

if project_name.is_dir():
    console.print(
        f"\nFolder [red]{project_name}[/red] exists. Please type another name\n",
    )
else:
    project_name.mkdir()
    new_cfg = {}
    new_cfg_path = Path.resolve(project_name / "manim.cfg")

    if not default_settings:
        for key, value in CFG_DEFAULTS.items():
            if key == "scene_names":
                new_cfg[key] = template_name + "Template"
            elif key == "resolution":
                new_cfg[key] = select_resolution()
            else:
                new_cfg[key] = click.prompt(f"\n{key}", default=value)

    console.print("\n", new_cfg)
    if click.confirm("Do you want to continue?", default=True, abort=True):
        copy_template_files(project_name, template_name)
        update_cfg(new_cfg, new_cfg_path)
    else:
        copy_template_files(project_name, template_name)
        update_cfg(CFG_DEFAULTS, new_cfg_path)

@cloup.command(
    context_settings=CONTEXT_SETTINGS,
    no_args_is_help=True,
    epilog=EPILOG,
)
@cloup.argument("scene_name", type=str, required=True)
@cloup.argument("file_name", type=str, required=False)
def scene(**args):
    """Inserts a SCENE to an existing FILE or creates a new FILE.

SCENE is the name of the scene that will be inserted.

FILE is the name of file in which the SCENE will be inserted.
"""
    template_name = click.prompt(
        "template",
        type=click.Choice(get_template_names(), False),
        default="Default",
    )
    scene = (get_template_path() / f"{template_name}.mtp").resolve().read_text()
    scene = scene.replace(template_name + "Template", args["scene_name"], 1)

    if args["file_name"]:
        file_name = Path(args["file_name"])

        if file_name.suffix != ".py":
            file_name = file_name.with_suffix(file_name.suffix + ".py")

        if file_name.is_file():
            # file exists so we are going to append new scene to that file
            with file_name.open("a") as f:
                f.write("\n\n\n" + scene)
        else:
            # file does not exist so we create a new file, append the scene and prepend the import statement
            file_name.write_text("\n\n\n" + scene)

            add_import_statement(file_name)
    else:
        # file name is not provided so we assume it is main.py
        # if main.py does not exist we do not continue
        with Path("main.py").open("a") as f:
            f.write("\n\n\n" + scene)

@cloup.group(
    context_settings=CONTEXT_SETTINGS,
    invoke_without_command=True,
    no_args_is_help=True,
    epilog=EPILOG,
    help="Create a new project or insert a new scene.",
)
@cloup.pass_context
def init(ctx):
    pass

init.add_command(project)
init.add_command(scene)

```

rotation.py

```
"""Animations related to rotation."""

from __future__ import annotations

__all__ = ["Rotating", "Rotate"]

from typing import TYPE_CHECKING, Callable, Sequence

import numpy as np

from ..animation.animation import Animation
from ..animation.transform import Transform
from ..constants import OUT, PI, TAU
from ..utils.rate_functions import linear

if TYPE_CHECKING:
    from ..mobject.mobject import Mobject


class Rotating(Animation):
    def __init__(self,
                 mobject: Mobject,
                 axis: np.ndarray = OUT,
                 radians: np.ndarray = TAU,
                 about_point: np.ndarray | None = None,
                 about_edge: np.ndarray | None = None,
                 run_time: float = 5,
                 rate_func: Callable[[float], float] = linear,
                 **kwargs,
                 ) -> None:
        self.axis = axis
        self.radians = radians
        self.about_point = about_point
        self.about_edge = about_edge
        super().__init__(mobject, run_time=run_time, rate_func=rate_func, **kwargs)

    def interpolate_mobject(self, alpha: float) -> None:
        self.mobject.become(self.starting_mobject)
        self.mobject.rotate(
            self.rate_func(alpha) * self.radians,
            axis=self.axis,
            about_point=self.about_point,
            about_edge=self.about_edge,
        )


class Rotate(Transform):
    """Animation that rotates a Mobject.

    Parameters
    -----
    mobject
        The mobject to be rotated.
    angle
        The rotation angle.
    axis
        The rotation axis as a numpy vector.
    about_point
        The rotation center.
    about_edge
        If ``about_point`` is ``None``, this argument specifies
        the direction of the bounding box point to be taken as
        the rotation center.

    Examples
    -----
    .. manim:: UsingRotate

        class UsingRotate(Scene):
            def construct(self):
                self.play(
                    Rotate(
                        Square(side_length=0.5).shift(UP * 2),
                        angle=2*PI,
                        about_point=ORIGIN,
                        rate_func=linear,
                    ),
                    Rotate(Square(side_length=0.5), angle=2*PI, rate_func=linear),
                )
    """

    def __init__(self,
                 mobject: Mobject,
                 angle: float,
                 about_point: np.ndarray | None = None,
                 about_edge: np.ndarray | None = None,
                 rate_func: Callable[[float], float] = linear,
                 run_time: float = 5,
                 **kwargs,
                 ) -> None:
        self.angle = angle
        self.about_point = about_point
        self.about_edge = about_edge
        self.rate_func = rate_func
        self.run_time = run_time
        super().__init__(mobject, **kwargs)
```

```
"""
def __init__(  
    self,  
    mobject: Mobject,  
    angle: float = PI,  
    axis: np.ndarray = OUT,  
    about_point: Sequence[float] | None = None,  
    about_edge: Sequence[float] | None = None,  
    **kwargs,  
) -> None:  
    if "path_arc" not in kwargs:  
        kwargs["path_arc"] = angle  
    if "path_arc_axis" not in kwargs:  
        kwargs["path_arc_axis"] = axis  
    self.angle = angle  
    self.axis = axis  
    self.about_edge = about_edge  
    self.about_point = about_point  
    if self.about_point is None:  
        self.about_point = mobject.get_center()  
    super().__init__(mobject, path_arc_centers=self.about_point, **kwargs)  
  
def create_target(self) -> Mobject:  
    target = self.mobject.copy()  
    target.rotate(  
        self.angle,  
        axis=self.axis,  
        about_point=self.about_point,  
        about_edge=self.about_edge,  
)  
    return target
```

debug.py

```
"""Debugging utilities."""

from __future__ import annotations
__all__ = ["print_family", "index_labels"]

from manim.mobject.mobject import Mobject
from manim.mobject.text.numbers import Integer

from ..mobject.types.vectorized_mobject import VGroup
from .color import BLACK

def print_family(mobject, n_tabs=0):
    """For debugging purposes"""
    print("\t" * n_tabs, mobject, id(mobject))
    for submob in mobject.submobjects:
        print_family(submob, n_tabs + 1)

def index_labels(
    mobject: Mobject,
    label_height: float = 0.15,
    background_stroke_width=5,
    background_stroke_color=BLACK,
    **kwargs,
) :
    r"""Returns a :class:`~.VGroup` of :class:`~.Integer` mobjects
    that shows the index of each submobject.

    Useful for working with parts of complicated mobjects.

    Parameters
    -----
    mobject
        The mobject that will have its submobjects labelled.
    label_height
        The height of the labels, by default 0.15.
    background_stroke_width
        The stroke width of the outline of the labels, by default 5.
    background_stroke_color
        The stroke color of the outline of labels.
    kwargs
        Additional parameters to be passed into the :class:`~.Integer`
        mobjects used to construct the labels.

    Examples
    -----
    .. manim:: IndexLabelsExample
        :save_last_frame:

        class IndexLabelsExample(Scene):
            def construct(self):
                text = MathTex(
                    "\frac{d}{dx}f(x)g(x) =",
                    "f(x)\frac{d}{dx}g(x)",
                    "+",
                    "g(x)\frac{d}{dx}f(x)",
                )

                #index the fist term in the MathTex mob
                indices = index_labels(text[0])

                text[0][1].set_color(PURPLE_B)
                text[0][8:12].set_color(DARK_BLUE)

                self.add(text, indices)
    """

    labels = VGroup()
    for n, submob in enumerate(mobject):
        label = Integer(n, **kwargs)
        label.set_stroke(
            background_stroke_color, background_stroke_width, background=True
        )
        label.height = label_height
        label.move_to(submob)
        labels.add(label)
    return labels
```


code_mobject.py

```
"""Mobject representing highlighted source code listings."""

from __future__ import annotations

__all__ = [
    "Code",
]

import html
import os
import re
from pathlib import Path

import numpy as np
from pygments import highlight
from pygments.formatters.html import HtmlFormatter
from pygments.lexers import get_lexer_by_name, guess_lexer_for_filename
from pygments.styles import get_all_styles

from manim import logger
from manim.constants import *
from manim.mobject.geometry.arc import Dot
from manim.mobject.geometry.polygram import RoundedRectangle
from manim.mobject.geometry.shape_matchers import SurroundingRectangle
from manim.mobject.text.text_mobject import Paragraph
from manim.mobject.types.vectorized_mobject import VGroup
from manim.utils.color import WHITE

class Code(VGroup):
    """A highlighted source code listing.

    An object ``listing`` of :class:`.Code` is a :class:`.VGroup` consisting
    of three objects:

    - The background, ``listing.background_mobject``. This is either
      a :class:`.Rectangle` (if the listing has been initialized with
      ``background="rectangle"``), the default option) or a :class:`.VGroup`
      resembling a window (if ``background="window"`` has been passed).

    - The line numbers, ``listing.line_numbers`` (a :class:`.Paragraph`
      object).

    - The highlighted code itself, ``listing.code`` (a :class:`.Paragraph`
      object).

    .. WARNING::

        Using a :class:`.Transform` on text with leading whitespace (and in
        this particular case: code) can look
        `weird <https://github.com/3b1b/manim/issues/1067>`. Consider using
        :meth:`remove_invisible_chars` to resolve this issue.

    Examples
    -------

    Normal usage::

        listing = Code(
            "helloworldcpp.cpp",
            tab_width=4,
            background_stroke_width=1,
            background_stroke_color=WHITE,
            insert_line_no=True,
            style=Code.styles_list[15],
            background="window",
            language="cpp",
        )

    We can also render code passed as a string (but note that
    the language has to be specified in this case)::

        .. manim:: CodeFromString
            :save_last_frame:

            class CodeFromString(Scene):
                def construct(self):
                    code = '''from manim import Scene, Square

class FadeInSquare(Scene):
    def construct(self):
        s = Square()
        self.play(FadeIn(s))
        self.play(s.animate.scale(2))
        self.wait()
'''
                    rendered_code = Code(code=code, tab_width=4, background="window",
                                         language="Python", font="Monospace")
                    self.add(rendered_code)

    Parameters
    -------

    file_name
        Name of the code file to display.
    code
        If ``file_name`` is not specified, a code string can be
        passed directly.
    tab_width
        Number of space characters corresponding to a tab character. Defaults to 3.
```

```

line_spacing
    Amount of space between lines in relation to font size. Defaults to 0.3, which means 30% of font size.
font_size
    A number which scales displayed code. Defaults to 24.
font
    The name of the text font to be used. Defaults to ``"Monospace"``.
    This is either a system font or one loaded with `text.register_font()`. Note
    that font family names may be different across operating systems.
stroke_width
    Stroke width for text. 0 is recommended, and the default.
margin
    Inner margin of text from the background. Defaults to 0.3.
indentation_chars
    "Indentation chars" refers to the spaces/tabs at the beginning of a given code line. Defaults to ``"    `` (spaces).
background
    Defines the background's type. Currently supports only ``"rectangle"`` (default) and ``"window"``.
background_stroke_width
    Defines the stroke width of the background. Defaults to 1.
background_stroke_color
    Defines the stroke color for the background. Defaults to ``WHITE``.
corner_radius
    Defines the corner radius for the background. Defaults to 0.2.
insert_line_no
    Defines whether line numbers should be inserted in displayed code. Defaults to ``True``.
line_no_from
    Defines the first line's number in the line count. Defaults to 1.
line_no_buff
    Defines the spacing between line numbers and displayed code. Defaults to 0.4.
style
    Defines the style type of displayed code. You can see possible names of styles in with :attr:`styles_list`. Defaults to ``"vim"``.
language
    Specifies the programming language the given code was written in. If ``None``
    (the default), the language will be automatically detected. For the list of
    possible options, visit https://pygments.org/docs/lexers/ and look for
    'aliases or short names'.
generate_html_file
    Defines whether to generate highlighted html code to the folder `assets/codes/generated_html_files`. Defaults to `False`.
warn_missing_font
    If True (default), Manim will issue a warning if the font does not exist in the
    (case-sensitive) list of fonts returned from `manimpango.list_fonts()`.

Attributes
-----
background_mobject : :class:`~.VGroup`
    The background of the code listing.
line_numbers : :class:`~.Paragraph`
    The line numbers for the code listing. Empty, if
    ``insert_line_no=False`` has been specified.
code : :class:`~.Paragraph`
    The highlighted code.

"""

# tuples in the form (name, aliases, filetypes, mimetypes)
# 'language' is aliases or short names
# For more information about pygments.lexers visit https://pygments.org/docs/lexers/
# from pygments.lexers import get_all_lexers
# all_lexers = get_all_lexers()
styles_list = list(get_all_styles())
# For more information about pygments.styles visit https://pygments.org/docs/styles/

def __init__(
    self,
    file_name: str | os.PathLike | None = None,
    code: str | None = None,
    tab_width: int = 3,
    line_spacing: float = 0.3,
    font_size: float = 24,
    font: str = "Monospace", # This should be in the font list on all platforms.
    stroke_width: float = 0,
    margin: float = 0.3,
    indentation_chars: str = "    ",
    background: str = "rectangle", # or window
    background_stroke_width: float = 1,
    background_stroke_color: str = WHITE,
    corner_radius: float = 0.2,
    insert_line_no: bool = True,
    line_no_from: int = 1,
    line_no_buff: float = 0.4,
    style: str = "vim",
    language: str | None = None,
    generate_html_file: bool = False,
    warn_missing_font: bool = True,
    **kwargs,
):
    super().__init__(
        stroke_width=stroke_width,
        **kwargs,
    )
    self.background_stroke_color = background_stroke_color
    self.background_stroke_width = background_stroke_width
    self.tab_width = tab_width
    self.line_spacing = line_spacing
    self.warn_missing_font = warn_missing_font
    self.font = font
    self.font_size = font_size
    self.margin = margin
    self.indentation_chars = indentation_chars
    self.background = background
    self.corner_radius = corner_radius
    self.insert_line_no = insert_line_no
    self.line_no_from = line_no_from
    self.line_no_buff = line_no_buff
    self.style = style
    self.language = language

```

```

self.generate_html_file = generate_html_file

self.file_path = None
self.file_name = file_name
if self.file_name:
    self._ensure_valid_file()
    self.code_string = self.file_path.read_text(encoding="utf-8")
elif code:
    self.code_string = code
else:
    raise ValueError(
        "Neither a code file nor a code string have been specified.",
    )
if isinstance(self.style, str):
    self.style = self.style.lower()
self._gen_html_string()
strati = self.html_string.find("background:")
self.background_color = self.html_string[strati + 12 : strati + 19]
self._gen_code_json()

self.code = self._gen_colored_lines()
if self.insert_line_no:
    self.line_numbers = self._gen_line_numbers()
    self.line_numbers.next_to(self.code, direction=LEFT, buff=self.line_no_buff)
if self.background == "rectangle":
    if self.insert_line_no:
        foreground = VGroup(self.code, self.line_numbers)
    else:
        foreground = self.code
    rect = SurroundingRectangle(
        foreground,
        buff=self.margin,
        color=self.background_color,
        fill_color=self.background_color,
        stroke_width=self.background_stroke_width,
        stroke_color=self.background_stroke_color,
        fill_opacity=1,
    )
    rect.round_corners(self.corner_radius)
    self.background_mobject = rect
else:
    if self.insert_line_no:
        foreground = VGroup(self.code, self.line_numbers)
    else:
        foreground = self.code
    height = foreground.height + 0.1 * 3 + 2 * self.margin
    width = foreground.width + 0.1 * 3 + 2 * self.margin

    rect = RoundedRectangle(
        corner_radius=self.corner_radius,
        height=height,
        width=width,
        stroke_width=self.background_stroke_width,
        stroke_color=self.background_stroke_color,
        color=self.background_color,
        fill_opacity=1,
    )
    red_button = Dot(radius=0.1, stroke_width=0, color="#ff5f56")
    red_button.shift(LEFT * 0.1 * 3)
    yellow_button = Dot(radius=0.1, stroke_width=0, color="#ffbd2e")
    green_button = Dot(radius=0.1, stroke_width=0, color="#27c93f")
    green_button.shift(RIGHT * 0.1 * 3)
    buttons = VGroup(red_button, yellow_button, green_button)
    buttons.shift(
        UP * (height / 2 - 0.1 * 2 - 0.05)
        + LEFT * (width / 2 - 0.1 * 5 - self.corner_radius / 2 - 0.05),
    )

    self.background_mobject = VGroup(rect, buttons)
    x = (height - foreground.height) / 2 - 0.1 * 3
    self.background_mobject.shift(foreground.get_center())
    self.background_mobject.shift(UP * x)
if self.insert_line_no:
    super().__init__(
        self.background_mobject, self.line_numbers, self.code, **kwargs
    )
else:
    super().__init__(
        self.background_mobject,
        Dot(fill_opacity=0, stroke_opacity=0),
        self.code,
        **kwargs,
    )
self.move_to(np.array([0, 0, 0]))

def _ensure_valid_file(self):
    """Function to validate file."""
    if self.file_name is None:
        raise Exception("Must specify file for Code")
    possible_paths = [
        Path() / "assets" / "codes" / self.file_name,
        Path(self.file_name).expanduser(),
    ]
    for path in possible_paths:
        if path.exists():
            self.file_path = path
            return
    error = (
        f"From: {Path.cwd()}, could not find {self.file_name} at either "
        + f"of these locations: {list(map(str, possible_paths))}"
    )
    raise OSError(error)

def _gen_line_numbers(self):
    """Function to generate line_numbers.

```

```

>Returns
-----
:class:`~.Paragraph`
    The generated line_numbers according to parameters.
"""
line_numbers_array = []
for line_no in range(0, self.code_json.__len__()):
    number = str(self.line_no_from + line_no)
    line_numbers_array.append(number)
line_numbers = Paragraph(
    *list(line_numbers_array),
    line_spacing=self.line_spacing,
    alignment="right",
    font_size=self.font_size,
    font=self.font,
    disable_ligatures=True,
    stroke_width=self.stroke_width,
    warn_missing_font=self.warn_missing_font,
)
for i in line_numbers:
    i.set_color(self.default_color)
return line_numbers

def _gen_colored_lines(self):
    """Function to generate code.

>Returns
-----
:class:`~.Paragraph`
    The generated code according to parameters.
"""
lines_text = []
for line_no in range(0, self.code_json.__len__()):
    line_str = ""
    for word_index in range(self.code_json[line_no].__len__()):
        line_str = line_str + self.code_json[line_no][word_index][0]
    lines_text.append(self.tab_spaces[line_no] * "\t" + line_str)
code = Paragraph(
    *list(lines_text),
    line_spacing=self.line_spacing,
    tab_width=self.tab_width,
    font_size=self.font_size,
    font=self.font,
    disable_ligatures=True,
    stroke_width=self.stroke_width,
    warn_missing_font=self.warn_missing_font,
)
for line_no in range(code.__len__()):
    line = code.chars[line_no]
    line_char_index = self.tab_spaces[line_no]
    for word_index in range(self.code_json[line_no].__len__()):
        line[
            line_char_index : line_char_index
            + self.code_json[line_no][word_index][0].__len__()
        ].set_color(self.code_json[line_no][word_index][1])
        line_char_index += self.code_json[line_no][word_index][0].__len__()
return code

def _gen_html_string(self):
    """Function to generate html string with code highlighted and stores in variable html_string."""
    self.html_string = _hilite_me(
        self.code_string,
        self.language,
        self.style,
        self.insert_line_no,
        "border:solid gray; border-width:.1em .1em .1em .8em; padding:.2em .6em;",
        self.file_path,
        self.line_no_from,
    )

    if self.generate_html_file:
        output_folder = Path() / "assets" / "codes" / "generated_html_files"
        output_folder.mkdir(parents=True, exist_ok=True)
        (output_folder / f"{self.file_name}.html").write_text(self.html_string)

def _gen_code_json(self):
    """Function to background_color, generate code_json and tab_spaces from html_string.
background_color is just background color of displayed code.
code_json is 2d array with rows as line numbers
and columns as a array with length 2 having text and text's color value.
tab_spaces is 2d array with rows as line numbers
and columns as corresponding number of indentation_chars in front of that line in code.
"""
    if (
        self.background_color == "#111111"
        or self.background_color == "#272822"
        or self.background_color == "#202020"
        or self.background_color == "#000000"
    ):
        self.default_color = "#ffffff"
    else:
        self.default_color = "#000000"
    # print(self.default_color, self.background_color)
    for i in range(3, -1, -1):
        self.html_string = self.html_string.replace("</> " * i, "</>")

    # handle pygments bug
    # https://github.com/pygments/pygments/issues/961
    self.html_string = self.html_string.replace("<span></span>", "")

    for i in range(10, -1, -1):
        self.html_string = self.html_string.replace(
            "</span>" + " " * i,
            " " * i + "</span>",

```

```

        )
self.html_string = self.html_string.replace("background-color:", "background:")

if self.insert_line_no:
    start_point = self.html_string.find("</td><td><pre"
    start_point = start_point + 9
else:
    start_point = self.html_string.find("<pre")
self.html_string = self.html_string[start_point:]
# print(self.html_string)
lines = self.html_string.split("\n")
lines = lines[0 : lines.__len__() - 2]
start_point = lines[0].find(">")
lines[0] = lines[0][start_point + 1 :]
# print(lines)
self.code_json = []
self.tab_spaces = []
code_json_line_index = -1
for line_index in range(0, lines.__len__()):
    # print(lines[line_index])
    self.code_json.append([])
    code_json_line_index = code_json_line_index + 1
    if lines[line_index].startswith(self.indentation_chars):
        start_point = lines[line_index].find("<")
        starting_string = lines[line_index][:start_point]
        indentation_chars_count = lines[line_index][:start_point].count(
            self.indentation_chars,
        )
    if (
        starting_string.__len__()
        != indentation_chars_count * self.indentation_chars.__len__()
    ):
        lines[line_index] = (
            "\t" * indentation_chars_count
            + starting_string[
                starting_string.rfind(self.indentation_chars)
                + self.indentation_chars.__len__() :
            ]
            + lines[line_index][start_point:]
        )
    else:
        lines[line_index] = (
            "\t" * indentation_chars_count + lines[line_index][start_point:]
        )
    indentation_chars_count = 0
    if lines[line_index]:
        while lines[line_index][indentation_chars_count] == "\t":
            indentation_chars_count = indentation_chars_count + 1
    self.tab_spaces.append(indentation_chars_count)
# print(lines[line_index])
lines[line_index] = self._correct_non_span(lines[line_index])
# print(lines[line_index])
words = lines[line_index].split("<span")
for word_index in range(1, words.__len__()):
    color_index = words[word_index].find("color:")
    if color_index == -1:
        color = self.default_color
    else:
        starti = words[word_index][color_index:].find("#")
        color = words[word_index][
            color_index + starti : color_index + starti + 7
        ]
    start_point = words[word_index].find(">")
    end_point = words[word_index].find("</span>")
    text = words[word_index][start_point + 1 : end_point]
    text = html.unescape(text)
    if text != "":
        # print(text, "" + color + "'")
        self.code_json[code_json_line_index].append([text, color])
# print(self.code_json)

def _correct_non_span(self, line_str: str):
    """Function put text color to those strings that don't have one according to background_color of displayed code.

Parameters
-----
line_str
    Takes a html element's string to put color to it according to background_color of displayed code.

Returns
-----
:class:`str`
    The generated html element's string with having color attributes.
"""
    words = line_str.split("</span>")
    line_str = ""
    for i in range(0, words.__len__()):
        if i != words.__len__() - 1:
            j = words[i].find("<span")
        else:
            j = words[i].__len__()
            temp = ""
            starti = -1
            for k in range(0, j):
                if words[i][k] == "\t" and starti == -1:
                    continue
                else:
                    if starti == -1:
                        starti = k
                    temp = temp + words[i][k]
    if temp != "":
        if i != words.__len__() - 1:
            temp = (
                '<span style="color:' +
                self.default_color

```

```

        + '>'
        + words[i][starti:j]
        + "</span>" )
    else:
        temp = (
            '<span style="color:'
            + self.default_color
            + '">'
            + words[i][starti:j]
        )
    temp = temp + words[i][j:]
    words[i] = temp
if words[i] != "":
    line_str = line_str + words[i] + "</span>"
return line_str

def _hilite_me(
    code: str,
    language: str,
    style: str,
    insert_line_no: bool,
    divstyles: str,
    file_path: Path,
    line_no_from: int,
):
    """Function to highlight code from string to html.

    Parameters
    -----
    code
        Code string.
    language
        The name of the programming language the given code was written in.
    style
        Code style name.
    insert_line_no
        Defines whether line numbers should be inserted in the html file.
    divstyles
        Some html css styles.
    file_path
        Path of code file.
    line_no_from
        Defines the first line's number in the line count.
    """
    style = style or "colorful"
    defstyles = "overflow:auto;width:auto;"

    formatter = HtmlFormatter(
        style=style,
        linenos=False,
        noclasses=True,
        cssclass="",
        cssstyles=defstyles + divstyles,
        prestyles="margin: 0",
    )
    if language is None and file_path:
        lexer = guess_lexer_for_filename(file_path, code)
        html = highlight(code, lexer, formatter)
    elif language is None:
        raise ValueError(
            "The code language has to be specified when rendering a code string",
        )
    else:
        html = highlight(code, get_lexer_by_name(language, **{}), formatter)
    if insert_line_no:
        html = _insert_line_numbers_in_html(html, line_no_from)
    html = "<!!-- HTML generated by Code() -->" + html
    return html

def _insert_line_numbers_in_html(html: str, line_no_from: int):
    """Function that inserts line numbers in the highlighted HTML code.

    Parameters
    -----
    html
        html string of highlighted code.
    line_no_from
        Defines the first line's number in the line count.

    Returns
    -----
    :class:`str`
        The generated html string with having line numbers.
    """
    match = re.search("<pre[^>]*>(.*)</pre>", html, re.DOTALL)
    if not match:
        return html
    pre_open = match.group(1)
    pre = match.group(2)
    pre_close = match.group(3)

    html = html.replace(pre_close, "</pre></td></tr></table>")
    numbers = range(line_no_from, line_no_from + pre.count("\n") + 1)
    format_lines = "%" + str(len(str(numbers[-1]))) + "i"
    lines = "\n".join(format_lines % i for i in numbers)
    html = html.replace(
        pre_open,
        "<table><tr><td>" + pre_open + lines + "</pre></td><td>" + pre_open,
    )
    return html

```

polygram.py

```
r"""Mobjects that are simple geometric shapes."""
from __future__ import annotations

__all__ = [
    "Polygram",
    "Polygon",
    "RegularPolygram",
    "RegularPolygon",
    "Star",
    "Triangle",
    "Rectangle",
    "Square",
    "RoundedRectangle",
    "Cutout",
]
from math import ceil
from typing import Iterable, Sequence

import numpy as np
from colour import Color

from manim.constants import *
from manim.mobject.geometry.arc import ArcBetweenPoints
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.color import *
from manim.utils.iterables import adjacent_n_tuples, adjacent_pairs
from manim.utils.space_ops import angle_between_vectors, normalize, regular_vertices

class Polygram(VMobject, metaclass=ConvertToOpenGL):
    """A generalized :class:`Polygon`, allowing for disconnected sets of edges.

    Parameters
    -----
    vertex_groups
        The groups of vertices making up the :class:`Polygram`.

        The first vertex in each group is repeated to close the shape.
        Each point must be 3-dimensional: ``[x,y,z]``

    color
        The color of the :class:`Polygram`.

    kwargs
        Forwarded to the parent constructor.

    Examples
    -----
    .. manim:: PolygramExample

        import numpy as np

        class PolygramExample(Scene):
            def construct(self):
                hexagram = Polygram(
                    [[0, 2, 0], [-np.sqrt(3), -1, 0], [np.sqrt(3), -1, 0]],
                    [[-np.sqrt(3), 1, 0], [0, -2, 0], [np.sqrt(3), 1, 0]],
                )
                self.add(hexagram)

                dot = Dot()
                self.play(MoveAlongPath(dot, hexagram), run_time=5, rate_func=linear)
                self.remove(dot)
                self.wait()

    """

    def __init__(self, *vertex_groups: Iterable[Sequence[float]], color=BLUE, **kwargs):
        super().__init__(color=color, **kwargs)

        for vertices in vertex_groups:
            first_vertex, *vertices = vertices
            first_vertex = np.array(first_vertex)

            self.start_new_path(first_vertex)
            self.add_points_as_corners([
                *(np.array(vertex) for vertex in vertices), first_vertex,
            ])

    def get_vertices(self) -> np.ndarray:
        """Gets the vertices of the :class:`Polygram`.
```

```

Returns
-----
:class:`numpy.ndarray`
    The vertices of the :class:`Polygram`.

Examples
-----
:::

>>> sq = Square()
>>> sq.get_vertices()
array([[ 1.,  1.,  0.],
       [-1.,  1.,  0.],
       [-1., -1.,  0.],
       [ 1., -1.,  0.]])
"""

return self.get_start_anchors()

def get_vertex_groups(self) -> np.ndarray:
    """Gets the vertex groups of the :class:`Polygram`.

    Returns
    -----
    :class:`numpy.ndarray`
        The vertex groups of the :class:`Polygram`.

    Examples
    -----
    :::

    >>> poly = Polygram([ORIGIN, RIGHT, UP], [LEFT, LEFT + UP, 2 * LEFT])
    >>> poly.get_vertex_groups()
    array([[ 0.,  0.,  0.],
           [ 1.,  0.,  0.],
           [ 0.,  1.,  0.],
           <BLANKLINE>
           [-1.,  0.,  0.],
           [-1.,  1.,  0.],
           [-2.,  0.,  0.]])
"""

vertex_groups = []

group = []
for start, end in zip(self.get_start_anchors(), self.get_end_anchors()):
    group.append(start)

    if self.consider_points_equals(end, group[0]):
        vertex_groups.append(group)
        group = []

return np.array(vertex_groups)

def round_corners(
    self,
    radius: float | list[float] = 0.5,
    evenly_distribute_anchors: bool = False,
    components_per_rounded_corner: int = 2,
):
    """Rounds off the corners of the :class:`Polygram`.

    Parameters
    -----
    radius
        The curvature of the corners of the :class:`Polygram`.
    evenly_distribute_anchors
        Break long line segments into proportionally-sized segments.
    components_per_rounded_corner
        The number of points used to represent the rounded corner curve.

    .. seealso::
        :class:`.~RoundedRectangle`

    .. note::
        If `radius` is supplied as a single value, then the same radius will be applied to all corners. If `radius` is a list, then the individual values will be applied sequentially, with the first corner receiving `radius[0]`, the second corner receiving `radius[1]`, etc. The radius list will be repeated as necessary.

    The `components_per_rounded_corner` value is provided so that the fidelity of the rounded corner may be fine-tuned as needed. 2 is an appropriate value for most shapes, however a larger value may be need if the rounded corner is particularly large. 2 is the minimum

```

number allowed, representing the start and end of the curve. 3 will result in a start, middle, and end point, meaning 2 curves will be generated.

The option to `evenly_distribute_anchors` is provided so that the line segments (the part part of each line remaining after rounding off the corners) can be subdivided to a density similar to that of the average density of the rounded corners. This may be desirable in situations in which an even distribution of curves is desired for use in later transformation animations. Be aware, though, that enabling this option can result in an object containing significantly more points than the original, especially when the rounded corner curves are small.

Examples

```
-----  
.. manim:: PolygramRoundCorners  
    :save_last_frame:  
  
    class PolygramRoundCorners(Scene):  
        def construct(self):  
            star = Star(outer_radius=2)  
  
            shapes = VGroup(star)  
            shapes.add(star.copy().round_corners(radius=0.1))  
            shapes.add(star.copy().round_corners(radius=0.25))  
  
            shapes.arrange(RIGHT)  
            self.add(shapes)  
    """  
  
    if radius == 0:  
        return self  
  
    new_points = []  
  
    for vertices in self.get_vertex_groups():  
        arcs = []  
  
        # Repeat the radius list as necessary in order to provide a radius  
        # for each vertex.  
        if isinstance(radius, (int, float)):  
            radius_list = [radius] * len(vertices)  
        else:  
            radius_list = radius * ceil(len(vertices) / len(radius))  
  
        for currentRadius, (v1, v2, v3) in zip(  
            radius_list, adjacent_n_tuples(vertices, 3)  
        ):  
            vect1 = v2 - v1  
            vect2 = v3 - v2  
            unit_vect1 = normalize(vect1)  
            unit_vect2 = normalize(vect2)  
  
            angle = angle_between_vectors(vect1, vect2)  
            # Negative radius gives concave curves  
            angle *= np.sign(currentRadius)  
  
            # Distance between vertex and start of the arc  
            cut_off_length = currentRadius * np.tan(angle / 2)  
  
            # Determines counterclockwise vs. clockwise  
            sign = np.sign(np.cross(vect1, vect2)[2])  
  
            arc = ArcBetweenPoints(  
                v2 - unit_vect1 * cut_off_length,  
                v2 + unit_vect2 * cut_off_length,  
                angle=sign * angle,  
                num_components=components_per_rounded_corner,  
            )  
            arcs.append(arc)  
  
    if evenly_distribute_anchors:  
        # Determine the average length of each curve  
        nonZeroLengthArcs = [arc for arc in arcs if len(arc.points) > 4]  
        if len(nonZeroLengthArcs):  
            totalArcLength = sum(  
                [arc.get_arc_length() for arc in nonZeroLengthArcs]  
            )  
            totalCurveCount = (  
                sum([len(arc.points) for arc in nonZeroLengthArcs]) / 4  
            )  
            averageLengthPerCurve = totalArcLength / totalCurveCount  
        else:  
            averageLengthPerCurve = 1  
  
    # To ensure that we loop through starting with last
```

```

arcs = [arcs[-1], *arcs[:-1]]
from manim.mobject.geometry.line import Line

for arc1, arc2 in adjacent_pairs(arcs):
    new_points.extend(arc1.points)

    line = Line(arc1.get_end(), arc2.get_start())

    # Make sure anchors are evenly distributed, if necessary
    if evenly_distribute_anchors:
        line.insert_n_curves(
            ceil(line.get_length() / averageLengthPerCurve)
        )

    new_points.extend(line.points)

self.set_points(new_points)

return self

class Polygon(Polygram):
    """A shape consisting of one closed loop of vertices.

    Parameters
    -----
    vertices
        The vertices of the :class:`Polygon`.
    kwargs
        Forwarded to the parent constructor.

    Examples
    -----
    .. manim:: PolygonExample
        :save_last_frame:

        class PolygonExample(Scene):
            def construct(self):
                isosceles = Polygon([-5, 1.5, 0], [-2, 1.5, 0], [-3.5, -2, 0])
                position_list = [
                    [4, 1, 0], # middle right
                    [4, -2.5, 0], # bottom right
                    [0, -2.5, 0], # bottom left
                    [0, 3, 0], # top left
                    [2, 1, 0], # middle
                    [4, 3, 0], # top right
                ]
                square_and_triangles = Polygon(*position_list, color=PURPLE_B)
                self.add(isosceles, square_and_triangles)
    """

    def __init__(self, *vertices: Sequence[float], **kwargs):
        super().__init__(vertices, **kwargs)

class RegularPolygram(Polygram):
    """A :class:`Polygram` with regularly spaced vertices.

    Parameters
    -----
    num_vertices
        The number of vertices.
    density
        The density of the :class:`RegularPolygram`.

        Can be thought of as how many vertices to hop
        to draw a line between them. Every ``density``-th
        vertex is connected.
    radius
        The radius of the circle that the vertices are placed on.
    start_angle
        The angle the vertices start at; the rotation of
        the :class:`RegularPolygram`.
    kwargs
        Forwarded to the parent constructor.

    Examples
    -----
    .. manim:: RegularPolygramExample
        :save_last_frame:

        class RegularPolygramExample(Scene):
            def construct(self):
                pentagram = RegularPolygram(5, radius=2)
                self.add(pentagram)
    """

```

```

def __init__(  

    self,  

    num_vertices: int,  

    *,  

    density: int = 2,  

    radius: float = 1,  

    start_angle: float | None = None,  

    **kwargs,  

):  

    # Regular polygrams can be expressed by the number of their vertices  

    # and their density. This relation can be expressed as its Schläfli  

    # symbol: {num_vertices/density}.  

    #  

    # For instance, a pentagon can be expressed as {5/1} or just {5}.  

    # A pentagram, however, can be expressed as {5/2}.  

    # A hexagram *would* be expressed as {6/2}, except that 6 and 2  

    # are not coprime, and it can be simplified to {3}, which corresponds  

    # to the fact that a hexagram is actually made up of 2 triangles.  

    #  

    # See https://en.wikipedia.org/wiki/Polygram\_\(geometry\)#Generalized\_regular\_polygons  

    # for more information.  

  

    num_gons = np.gcd(num_vertices, density)  

    num_vertices //= num_gons  

    density //= num_gons  

  

    # Utility function for generating the individual  

    # polygon vertices.  

    def gen_polygon_vertices(start_angle):  

        reg_vertices, start_angle = regular_vertices(  

            num_vertices,  

            radius=radius,  

            start_angle=start_angle,  

        )  

  

        vertices = []  

        i = 0  

        while True:  

            vertices.append(reg_vertices[i])  

  

            i += density  

            i %= num_vertices  

            if i == 0:  

                break  

  

        return vertices, start_angle  

  

    first_group, self.start_angle = gen_polygon_vertices(start_angle)  

    vertex_groups = [first_group]  

  

    for i in range(1, num_gons):  

        start_angle = self.start_angle + (i / num_gons) * TAU / num_vertices  

        group, _ = gen_polygon_vertices(start_angle)  

  

        vertex_groups.append(group)  

  

    super().__init__(*vertex_groups, **kwargs)  

  

class RegularPolygon(RegularPolygram):  

    """An n-sided regular :class:`Polygon`.  

  

    Parameters  

    -----  

    n  

        The number of sides of the :class:`RegularPolygon`.  

    kwargs  

        Forwarded to the parent constructor.  

  

    Examples  

    -----  

    .. manim:: RegularPolygonExample  

       :save_last_frame:  

  

       class RegularPolygonExample(Scene):  

           def construct(self):  

               poly_1 = RegularPolygon(n=6)  

               poly_2 = RegularPolygon(n=6, start_angle=30*DEGREES, color=GREEN)  

               poly_3 = RegularPolygon(n=10, color=RED)  

  

               poly_group = Group(poly_1, poly_2, poly_3).scale(1.5).arrange(buff=1)  

               self.add(poly_group)  

    """
  

  

def __init__(self, n: int = 6, **kwargs):  

    super().__init__(n, density=1, **kwargs)

```

```

class Star(Polygon):
    """A regular polygram without the intersecting lines.

Parameters
-----
n
    How many points on the :class:`Star`.
outer_radius
    The radius of the circle that the outer vertices are placed on.
inner_radius
    The radius of the circle that the inner vertices are placed on.

    If unspecified, the inner radius will be calculated such that the edges of the :class:`Star` perfectly follow the edges of its :class:`RegularPolygram` counterpart.
density
    The density of the :class:`Star`. Only used if ``inner_radius`` is unspecified.

    See :class:`RegularPolygram` for more information.
start_angle
    The angle the vertices start at; the rotation of the :class:`Star`.
kwargs
    Forwarded to the parent constructor.

Raises
-----
:exc:`ValueError`
    If ``inner_radius`` is unspecified and ``density`` is not in the range ``[1, n/2)``.

Examples
-----
.. manim:: StarExample
    :save_as_gif:

    class StarExample(Scene):
        def construct(self):
            pentagram = RegularPolygram(5, radius=2)
            star = Star(outer_radius=2, color=RED)

            self.add(pentagram)
            self.play(Create(star), run_time=3)
            self.play(FadeOut(star), run_time=2)

.. manim:: DifferentDensitiesExample
    :save_last_frame:

    class DifferentDensitiesExample(Scene):
        def construct(self):
            density_2 = Star(7, outer_radius=2, density=2, color=RED)
            density_3 = Star(7, outer_radius=2, density=3, color=PURPLE)

            self.add(VGroup(density_2, density_3).arrange(RIGHT))

"""

def __init__(
    self,
    n: int = 5,
    *,
    outer_radius: float = 1,
    inner_radius: float | None = None,
    density: int = 2,
    start_angle: float | None = TAU / 4,
    **kwargs,
):
    inner_angle = TAU / (2 * n)

    if inner_radius is None:
        # See https://math.stackexchange.com/a/2136292 for an
        # overview of how to calculate the inner radius of a
        # perfect star.

        if density <= 0 or density >= n / 2:
            raise ValueError(
                f"Incompatible density {density} for number of points {n}",
            )

    outer_angle = TAU * density / n
    inverse_x = 1 - np.tan(inner_angle) * (
        np.cos(outer_angle) - 1) / np.sin(outer_angle)
)

```

```

        inner_radius = outer_radius / (np.cos(inner_angle) * inverse_x)

    outer_vertices, self.start_angle = regular_vertices(
        n,
        radius=outer_radius,
        start_angle=start_angle,
    )
    inner_vertices, _ = regular_vertices(
        n,
        radius=inner_radius,
        start_angle=self.start_angle + inner_angle,
    )

    vertices = []
    for pair in zip(outer_vertices, inner_vertices):
        vertices.append(pair)

    super().__init__(vertices, **kwargs)

class Triangle(RegularPolygon):
    """An equilateral triangle.

    Parameters
    -----
    kwargs
        Additional arguments to be passed to :class:`RegularPolygon`

    Examples
    -----
    .. manim:: TriangleExample
        :save_last_frame:

        class TriangleExample(Scene):
            def construct(self):
                triangle_1 = Triangle()
                triangle_2 = Triangle().scale(2).rotate(60*DEGREES)
                tri_group = Group(triangle_1, triangle_2).arrange(buff=1)
                self.add(tri_group)
    """

    def __init__(self, **kwargs):
        super().__init__(n=3, **kwargs)

class Rectangle(Polygon):
    """A quadrilateral with two sets of parallel sides.

    Parameters
    -----
    color
        The color of the rectangle.
    height
        The vertical height of the rectangle.
    width
        The horizontal width of the rectangle.
    grid_xstep
        Space between vertical grid lines.
    grid_ystep
        Space between horizontal grid lines.
    mark_paths_closed
        No purpose.
    close_new_points
        No purpose.
    kwargs
        Additional arguments to be passed to :class:`Polygon`

    Examples
    -----
    .. manim:: RectangleExample
        :save_last_frame:

        class RectangleExample(Scene):
            def construct(self):
                rect1 = Rectangle(width=4.0, height=2.0, grid_xstep=1.0, grid_ystep=0.5)
                rect2 = Rectangle(width=1.0, height=4.0)

                rects = Group(rect1, rect2).arrange(buff=1)
                self.add(rects)
    """

    def __init__(
        self,
        color: Color = WHITE,
        height: float = 2.0,
        width: float = 4.0,
        grid_xstep: float | None = None,

```

```

        grid_ystep: float | None = None,
        mark_paths_closed: bool = True,
        close_new_points: bool = True,
        **kwargs,
    ):

        super().__init__(UR, UL, DL, DR, color=color, **kwargs)
        self.stretch_to_fit_width(width)
        self.stretch_to_fit_height(height)
        v = self.get_vertices()
        if grid_xstep is not None:
            from manim.mobject.geometry.line import Line

            grid_xstep = abs(grid_xstep)
            count = int(width / grid_xstep)
            grid = VGroup(
                *(
                    Line(
                        v[1] + i * grid_xstep * RIGHT,
                        v[1] + i * grid_xstep * RIGHT + height * DOWN,
                        color=color,
                    )
                    for i in range(1, count)
                )
            )
            self.add(grid)

        if grid_ystep is not None:
            grid_ystep = abs(grid_ystep)
            count = int(height / grid_ystep)
            grid = VGroup(
                *(
                    Line(
                        v[1] + i * grid_ystep * DOWN,
                        v[1] + i * grid_ystep * DOWN + width * RIGHT,
                        color=color,
                    )
                    for i in range(1, count)
                )
            )
            self.add(grid)

```

```

class Square(Rectangle):
    """A rectangle with equal side lengths.

    Parameters
    -----
    side_length
        The length of the sides of the square.
    kwargs
        Additional arguments to be passed to :class:`Rectangle`.

    Examples
    -----
    .. manim:: SquareExample
        :save_last_frame:

        class SquareExample(Scene):
            def construct(self):
                square_1 = Square(side_length=2.0).shift(DOWN)
                square_2 = Square(side_length=1.0).next_to(square_1, direction=UP)
                square_3 = Square(side_length=0.5).next_to(square_2, direction=UP)
                self.add(square_1, square_2, square_3)
    """

```

```

def __init__(self, side_length: float = 2.0, **kwargs):
    self.side_length = side_length
    super().__init__(height=side_length, width=side_length, **kwargs)

```

```

class RoundedRectangle(Rectangle):
    """A rectangle with rounded corners.

    Parameters
    -----
    corner_radius
        The curvature of the corners of the rectangle.
    kwargs
        Additional arguments to be passed to :class:`Rectangle`"

    Examples
    -----
    .. manim:: RoundedRectangleExample
        :save_last_frame:

        class RoundedRectangleExample(Scene):
            def construct(self):
                rect_1 = RoundedRectangle(corner_radius=0.5)

```

```

rect_2 = RoundedRectangle(corner_radius=1.5, height=4.0, width=4.0)

rect_group = Group(rect_1, rect_2).arrange(buff=1)
self.add(rect_group)
"""

def __init__(self, corner_radius: float | list[float] = 0.5, **kwargs):
    super().__init__(**kwargs)
    self.corner_radius = corner_radius
    self.round_corners(self.corner_radius)

class Cutout(VMobject, metaclass=ConvertToOpenGL):
    """A shape with smaller cutouts.

Parameters
-----
main_shape
    The primary shape from which cutouts are made.
mobjects
    The smaller shapes which are to be cut out of the ``main_shape``.
kwargs
    Further keyword arguments that are passed to the constructor of
    :class:`~.VMobject`.

.. warning::
    Technically, this class behaves similar to a symmetric difference: if
    parts of the ``mobjects`` are not located within the ``main_shape``,
    these parts will be added to the resulting :class:`~.VMobject`.

Examples
-----
.. manim:: CutoutExample

    class CutoutExample(Scene):
        def construct(self):
            s1 = Square().scale(2.5)
            s2 = Triangle().shift(DOWN + RIGHT).scale(0.5)
            s3 = Square().shift(UP + RIGHT).scale(0.5)
            s4 = RegularPolygon(5).shift(DOWN + LEFT).scale(0.5)
            s5 = RegularPolygon(6).shift(UP + LEFT).scale(0.5)
            c = Cutout(s1, s2, s3, s4, s5, fill_opacity=1, color=BLUE, stroke_color=RED)
            self.play(Write(c), run_time=4)
            self.wait()
"""

def __init__(self, main_shape: VMobject, *mobjects: VMobject, **kwargs):
    super().__init__(**kwargs)
    self.append_points(main_shape.points)
    if main_shape.get_direction() == "CW":
        sub_direction = "CCW"
    else:
        sub_direction = "CW"
    for mobject in mobjects:
        self.append_points(mobject.force_direction(sub_direction).points)

```

exceptions.py

```
from __future__ import annotations

class EndSceneEarlyException(Exception):
    pass

class RerunSceneException(Exception):
    pass

class MultiAnimationOverrideException(Exception):
    pass
```

unit.py

```
"""Implement the Unit class."""

from __future__ import annotations

import numpy as np

from .. import config, constants

__all__ = ["Pixels", "Degrees", "Munits", "Percent"]

class _PixelUnits:
    def __mul__(self, val):
        return val * config.frame_width / config.pixel_width

    def __rmul__(self, val):
        return val * config.frame_width / config.pixel_width

class Percent:
    def __init__(self, axis):
        if np.array_equal(axis, constants.X_AXIS):
            self.length = config.frame_width
        if np.array_equal(axis, constants.Y_AXIS):
            self.length = config.frame_height
        if np.array_equal(axis, constants.Z_AXIS):
            raise NotImplementedError("length of Z axis is undefined")

    def __mul__(self, val):
        return val / 100 * self.length

    def __rmul__(self, val):
        return val / 100 * self.length

Pixels = _PixelUnits()
Degrees = constants.PI / 180
Munits = 1
```

speedmodifier.py

```
"""Utilities for modifying the speed at which animations are played."""

from __future__ import annotations

import types
from typing import Callable

from numpy import piecewise

from manim.utils.simple_functions import get_parameters

from ..animation.animation import Animation, Wait, prepare_animation
from ..animation.composition import AnimationGroup
from ..mobject.mobject import Mobject, Updater, _AnimationBuilder
from ..scene.scene import Scene


class ChangeSpeed(Animation):
    """Modifies the speed of passed animation.
    :class:`AnimationGroup` with different ``lag_ratio`` can also be used
    which combines multiple animations into one.
    The ``run_time`` of the passed animation is changed to modify the speed.

    Parameters
    -----
    anim
        Animation of which the speed is to be modified.
    speedinfo
        Contains nodes (percentage of ``run_time``) and its corresponding speed factor.
    rate_func
        Overrides ``rate_func`` of passed animation, applied before changing speed.

    Examples
    -----
    .. manim:: SpeedModifierExample

        class SpeedModifierExample(Scene):
            def construct(self):
                a = Dot().shift(LEFT * 4)
                b = Dot().shift(RIGHT * 4)
                self.add(a, b)
                self.play(
                    ChangeSpeed(
                        AnimationGroup(
                            a.animate(run_time=1).shift(RIGHT * 8),
                            b.animate(run_time=1).shift(LEFT * 8),
                        ),
                        speedinfo={0.3: 1, 0.4: 0.1, 0.6: 0.1, 1: 1},
                        rate_func=linear,
                    )
                )

    .. manim:: SpeedModifierUpdaterExample

        class SpeedModifierUpdaterExample(Scene):
            def construct(self):
                a = Dot().shift(LEFT * 4)
                self.add(a)

                ChangeSpeed.add_updater(a, lambda x, dt: x.shift(RIGHT * 4 * dt))
                self.play(
                    ChangeSpeed(
                        Wait(2),
                        speedinfo={0.4: 1, 0.5: 0.2, 0.8: 0.2, 1: 1},
                        affects_speed_updaters=True,
                    )
                )

    .. manim:: SpeedModifierUpdaterExample2

        class SpeedModifierUpdaterExample2(Scene):
            def construct(self):
                a = Dot().shift(LEFT * 4)
                self.add(a)

                ChangeSpeed.add_updater(a, lambda x, dt: x.shift(RIGHT * 4 * dt))
                self.wait()
                self.play(
                    ChangeSpeed(
                        Wait(),

```

```

        speedinfo={1: 0},
        affects_speed_updaters=True,
    )
)

"""
dt = 0
is_changing_dt = False

def __init__(
    self,
    anim: Animation | _AnimationBuilder,
    speedinfo: dict[float, float],
    rate_func: Callable[[float], float] | None = None,
    affects_speed_updaters: bool = True,
    **kwargs,
) -> None:
    if issubclass(type(anim), AnimationGroup):
        self.anim = type(anim)(
            *map(self.setup, anim.animations),
            group=anim.group,
            run_time=anim.run_time,
            rate_func=anim.rate_func,
            lag_ratio=anim.lag_ratio,
        )
    else:
        self.anim = self.setup(anim)

    if affects_speed_updaters:
        assert (
            ChangeSpeed.is_changing_dt is False
        ), "Only one animation at a time can play that changes speed (dt) for ChangeSpeed updaters"
        ChangeSpeed.is_changing_dt = True
        self.t = 0
    self.affects_speed_updaters = affects_speed_updaters

    self.rate_func = self.anim.rate_func if rate_func is None else rate_func

    # A function where, f(0) = 0, f'(0) = initial speed, f'(f-1(1)) = final speed
    # Following function obtained when conditions applied to vertical parabola
    self.speed_modifier = lambda x, init_speed, final_speed: (
        (final_speed**2 - init_speed**2) * x**2 / 4 + init_speed * x
    )

    # f-1(1), returns x for which f(x) = 1 in `speed_modifier` function
    self.f_inv_1 = lambda init_speed, final_speed: 2 / (init_speed + final_speed)

    # if speed factors for the starting node (0) and the final node (1) are
    # not set, set them to 1 and the penultimate factor, respectively
    if 0 not in speedinfo:
        speedinfo[0] = 1
    if 1 not in speedinfo:
        speedinfo[1] = sorted(speedinfo.items())[-1][1]

    self.speedinfo = dict(sorted(speedinfo.items()))
    self.functions = []
    self.conditions = []

    # Get the time taken by amimation if `run_time` is assumed to be 1
    scaled_total_time = self.get_scaled_total_time()

    prevnode = 0
    init_speed = self.speedinfo[0]
    curr_time = 0
    for node, final_speed in list(self.speedinfo.items())[1:]:
        dur = node - prevnode

        def condition(
            t,
            curr_time=curr_time,
            init_speed=init_speed,
            final_speed=final_speed,
            dur=dur,
        ):
            lower_bound = curr_time / scaled_total_time
            upper_bound = (
                curr_time + self.f_inv_1(init_speed, final_speed) * dur
            ) / scaled_total_time
            return lower_bound <= t <= upper_bound

        self.conditions.append(condition)

        def function(
            t,
            curr_time=curr_time,
            init_speed=init_speed,
        ):
            if t <= lower_bound:
                return init_speed
            if t >= upper_bound:
                return final_speed
            x = (t - lower_bound) / dur
            return self.speed_modifier(x, init_speed, final_speed)

```

```

        final_speed=final_speed,
        dur=dur,
        prevnode=prevnode,
    ):

        return (
            self.speed_modifier(
                (scaled_total_time * t - curr_time) / dur,
                init_speed,
                final_speed,
            )
            * dur
            + prevnode
        )

    self.functions.append(function)

curr_time += self.f_inv_1(init_speed, final_speed) * dur
prevnode = node
init_speed = final_speed

def func(t):
    if t == 1:
        ChangeSpeed.is_changing_dt = False
    new_t = piecewise(
        self.rate_func(t),
        [condition(self.rate_func(t)) for condition in self.conditions],
        self.functions,
    )
    if self.affects_speed_updaters:
        ChangeSpeed.dt = (new_t - self.t) * self.anim.run_time
        self.t = new_t
    return new_t

self.anim.set_rate_func(func)

super().__init__(
    self.anim.mobject,
    rate_func=self.rate_func,
    run_time=scaled_total_time * self.anim.run_time,
    **kwargs,
)

def setup(self, anim):
    if type(anim) is Wait:
        anim.interpolate = types.MethodType(
            lambda self, alpha: self.rate_func(alpha), anim
        )
    return prepare_animation(anim)

def get_scaled_total_time(self) -> float:
    """The time taken by the animation under the assumption that the ``run_time`` is 1."""
    prevnode = 0
    init_speed = self.speedinfo[0]
    total_time = 0
    for node, final_speed in list(self.speedinfo.items())[1:]:
        dur = node - prevnode
        total_time += dur * self.f_inv_1(init_speed, final_speed)
        prevnode = node
        init_speed = final_speed
    return total_time

@classmethod
def add_updater(
    cls,
    mobject: Mobject,
    update_function: Updater,
    index: int | None = None,
    call_updater: bool = False,
):
    """This static method can be used to apply speed change to updaters.

    This updater will follow speed and rate function of any :class:`.ChangeSpeed` animation that is playing with ``affects_speed_updaters=True``. By default, updater functions added via the usual :meth:`.Mobject.add_updater` method do not respect the change of animation speed.

    Parameters
    -----
    mobject
        The mobject to which the updater should be attached.
    update_function
        The function that is called whenever a new frame is rendered.
    index
        The position in the list of the mobject's updaters at which the function should be inserted.
    call_updater
        If ``True``, calls the update function when attaching it to the

```

```
mobject.

See also
-----
:class:`.ChangeSpeed`
:meth:`.Mobject.add_updater`

parameters = get_parameters(update_function)
if "dt" in parameters:
    mobject.add_updater(
        lambda mob, dt: update_function(
            mob, ChangeSpeed.dt if ChangeSpeed.is_changing_dt else dt
        ),
        index=index,
        call_updater=call_updater,
    )
else:
    mobject.add_updater(update_function, index=index, call_updater=call_updater)

def interpolate(self, alpha: float) -> None:
    self.anim.interpolate(alpha)

def update_mobjects(self, dt: float) -> None:
    self.anim.update_mobjects(dt)

def finish(self) -> None:
    ChangeSpeed.is_changing_dt = False
    self.anim.finish()

def begin(self) -> None:
    self.anim.begin()

def clean_up_from_scene(self, scene: Scene) -> None:
    self.anim.clean_up_from_scene(scene)

def _setup_scene(self, scene) -> None:
    self.anim._setup_scene(scene)
```

tips.py

```

class ArrowTipsShowcase(Scene):
    def construct(self):
        a00 = Arrow(start=[-2, 3, 0], end=[2, 3, 0], color=WHITE)
        a11 = Arrow(start=[-2, 2, 0], end=[2, 2, 0], tip_shape=ArrowTriangleTip)
        a12 = Arrow(start=[-2, 1, 0], end=[2, 1, 0])
        a21 = Arrow(start=[-2, 0, 0], end=[2, 0, 0], tip_shape=ArrowSquareTip)
        a22 = Arrow([-2, -1, 0], [2, -1, 0], tip_shape=ArrowSquareFilledTip)
        a31 = Arrow([-2, -2, 0], [2, -2, 0], tip_shape=ArrowCircleTip)
        a32 = Arrow([-2, -3, 0], [2, -3, 0], tip_shape=ArrowCircleFilledTip)
        b11 = a11.copy().scale(0.5, scale_tips=True).next_to(a11, RIGHT)
        b12 = a12.copy().scale(0.5, scale_tips=True).next_to(a12, RIGHT)
        b21 = a21.copy().scale(0.5, scale_tips=True).next_to(a21, RIGHT)
        self.add(a00, a11, a12, a21, a22, a31, a32, b11, b12, b21)

    """
    def __init__(self, *args, **kwargs):
        raise NotImplementedError("Has to be implemented in inheriting subclasses.")

    @property
    def base(self):
        """The base point of the arrow tip.

        This is the point connecting to the arrow line.

        Examples
        -----
        ::

            >>> from manim import Arrow
            >>> arrow = Arrow(np.array([0, 0, 0]), np.array([2, 0, 0]), buff=0)
            >>> arrow.tip.base.round(2) + 0. # add 0. to avoid negative 0 in output
            array([1.65, 0., 0.])

        """
        return self.point_from_proportion(0.5)

    @property
    def tip_point(self):
        """The tip point of the arrow tip.

        Examples
        -----
        ::

            >>> from manim import Arrow
            >>> arrow = Arrow(np.array([0, 0, 0]), np.array([2, 0, 0]), buff=0)
            >>> arrow.tip.tip_point.round(2) + 0.
            array([2., 0., 0.])

        """
        return self.points[0]

    @property
    def vector(self):
        """The vector pointing from the base point to the tip point.

        Examples
        -----
        ::

            >>> from manim import Arrow
            >>> arrow = Arrow(np.array([0, 0, 0]), np.array([2, 2, 0]), buff=0)
            >>> arrow.tip.vector.round(2) + 0.
            array([0.25, 0.25, 0.])

        """
        return self.tip_point - self.base

    @property
    def tip_angle(self):
        """The angle of the arrow tip.

        Examples
        -----
        ::

            >>> from manim import Arrow
            >>> arrow = Arrow(np.array([0, 0, 0]), np.array([1, 1, 0]), buff=0)
            >>> round(arrow.tip.tip_angle, 5) == round(PI/4, 5)
            True

        """
        return angle_of_vector(self.vector)

    @property
    def length(self):

```

```

r"""The length of the arrow tip.

Examples
-----
::

    >>> from manim import Arrow
    >>> arrow = Arrow(np.array([0, 0, 0]), np.array([1, 2, 0]))
    >>> round(arrow.tip.length, 3)
    0.35

"""

return np.linalg.norm(self.vector)

class StealthTip(ArrowTip):
    r"""'Stealth' fighter / kite arrow shape.

    Naming is inspired by the corresponding
    `TikZ arrow shape <https://tikz.dev/tikz-arrows#sec-16.3>`__.

"""

    def __init__(self,
                 fill_opacity=1,
                 stroke_width=3,
                 length=DEFAULT_ARROW_TIP_LENGTH / 2,
                 start_angle=PI,
                 **kwargs,
                 ):
        self.start_angle = start_angle
        VMobObject.__init__(self, fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs)
        self.set_points_as_corners([
            [2, 0, 0], # tip
            [-1.2, 1.6, 0],
            [0, 0, 0], # base
            [-1.2, -1.6, 0],
            [2, 0, 0], # close path, back to tip
        ])
        self.scale(length / self.length)

    @property
    def length(self):
        """The length of the arrow tip.

        In this case, the length is computed as the height of
        the triangle encompassing the stealth tip (otherwise,
        the tip is scaled too large).
        """

        return np.linalg.norm(self.vector) * 1.6

class ArrowTriangleTip(ArrowTip, Triangle):
    r"""Triangular arrow tip."""

    def __init__(self,
                 fill_opacity=0,
                 stroke_width=3,
                 length=DEFAULT_ARROW_TIP_LENGTH,
                 width=DEFAULT_ARROW_TIP_LENGTH,
                 start_angle=PI,
                 **kwargs,
                 ):
        Triangle.__init__(self,
                          fill_opacity=fill_opacity,
                          stroke_width=stroke_width,
                          start_angle=start_angle,
                          **kwargs,
                          )
        self.width = width

        self.stretch_to_fit_width(length)
        self.stretch_to_fit_height(width)

class ArrowTriangleFilledTip(ArrowTriangleTip):
    r"""Triangular arrow tip with filled tip.

    This is the default arrow tip shape.
"""

```

```

def __init__(self, fill_opacity=1, stroke_width=0, **kwargs):
    super().__init__(fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs)

class ArrowCircleTip(ArrowTip, Circle):
    r"""Circular arrow tip."""

    def __init__(
        self,
        fill_opacity=0,
        stroke_width=3,
        length=DEFAULT_ARROW_TIP_LENGTH,
        start_angle=PI,
        **kwargs,
    ):
        self.start_angle = start_angle
        Circle.__init__(
            self, fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs
        )
        self.width = length
        self.stretch_to_fit_height(length)

class ArrowCircleFilledTip(ArrowCircleTip):
    r"""Circular arrow tip with filled tip."""

    def __init__(self, fill_opacity=1, stroke_width=0, **kwargs):
        super().__init__(fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs)

class ArrowSquareTip(ArrowTip, Square):
    r"""Square arrow tip."""

    def __init__(
        self,
        fill_opacity=0,
        stroke_width=3,
        length=DEFAULT_ARROW_TIP_LENGTH,
        start_angle=PI,
        **kwargs,
    ):
        self.start_angle = start_angle
        Square.__init__(
            self,
            fill_opacity=fill_opacity,
            stroke_width=stroke_width,
            side_length=length,
            **kwargs,
        )
        self.width = length
        self.stretch_to_fit_height(length)

class ArrowSquareFilledTip(ArrowSquareTip):
    r"""Square arrow tip with filled tip."""

    def __init__(self, fill_opacity=1, stroke_width=0, **kwargs):
        super().__init__(fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs)

```

utils.py_1

```
"""Utilities for working with mobjects."""

from __future__ import annotations

__all__ = [
    "get_mobject_class",
    "get_point_mobject_class",
    "get_vectorized_mobject_class",
]

from ..config import config
from ..constants import RendererType
from .mobject import Mobject
from .opengl.opengl_mobject import OpenGLMobject
from .opengl.opengl_point_cloud_mobject import OpenGLPobject
from .opengl.opengl_vectorized_mobject import OpenGLVMobject
from .types.point_cloud_mobject import PMobject
from .types.vectorized_mobject import VMobject

def get_mobject_class() -> type:
    """Gets the base mobject class, depending on the currently active renderer.

    .. NOTE::

        This method is intended to be used in the code base of Manim itself
        or in plugins where code should work independent of the selected
        renderer.

    Examples
    -------

    The function has to be explicitly imported. We test that
    the name of the returned class is one of the known mobject
    base classes:::

        >>> from manim.mobject.utils import get_mobject_class
        >>> get_mobject_class().__name__ in ['Mobject', 'OpenGLMobject']
        True
    """
    if config.renderer == RendererType.CAIRO:
        return Mobject
    if config.renderer == RendererType.OPENGL:
        return OpenGLMobject
    raise NotImplementedError(
        "Base mobjects are not implemented for the active renderer."
    )

def get_vectorized_mobject_class() -> type:
    """Gets the vectorized mobject class, depending on the currently
    active renderer.

    .. NOTE::

        This method is intended to be used in the code base of Manim itself
        or in plugins where code should work independent of the selected
        renderer.

    Examples
    -------

    The function has to be explicitly imported. We test that
    the name of the returned class is one of the known mobject
    base classes:::

        >>> from manim.mobject.utils import get_vectorized_mobject_class
        >>> get_vectorized_mobject_class().__name__ in ['VMobject', 'OpenGLVMobject']
        True
    """
    if config.renderer == RendererType.CAIRO:
        return VMobject
    if config.renderer == RendererType.OPENGL:
        return OpenGLVMobject
    raise NotImplementedError(
        "Vectorized mobjects are not implemented for the active renderer."
    )

def get_point_mobject_class() -> type:
    """Gets the point cloud mobject class, depending on the currently
```

```
active renderer.

.. NOTE::

This method is intended to be used in the code base of Manim itself
or in plugins where code should work independent of the selected
renderer.

Examples
-----
The function has to be explicitly imported. We test that
the name of the returned class is one of the known mobject
base classes::

>>> from manim.mobject.utils import get_point_mobject_class
>>> get_point_mobject_class().__name__ in ['PMobject', 'OpenGLMobject']
True
"""
if config.renderer == RendererType.CAIRO:
    return PMobject
if config.renderer == RendererType.OPENGL:
    return OpenGLMobject
raise NotImplementedError(
    "Point cloud mobjects are not implemented for the active renderer."
)
```

transform.py

```

        path.set_points_smoothly(points)
        return path

    left = Circle(stroke_color=BLUE_E, fill_opacity=1.0, radius=0.5).move_to(LEFT * 2)
    colors = [TEAL_A, TEAL_B, TEAL_C, TEAL_D, TEAL_E, GREEN_A]
    # Positive angles move counter-clockwise, negative angles move clockwise.
    examples = [-90, 0, 30, 90, 180, 270]
    anims = []
    for idx, angle in enumerate(examples):
        left_c = left.copy().shift((3 - idx) * UP)
        left_c.fill_color = colors[idx]
        right_c = left_c.copy().shift(4 * RIGHT)
        path_arc = make_arc_path(left_c.get_center(), right_c.get_center(),
                                 arc_angle=angle * DEGREES)
        desc = Text('%d°' % examples[idx]).next_to(left_c, LEFT)
        # Make the circles in front of the text in front of the arcs.
        self.add(
            path_arc.set_z_index(1),
            desc.set_z_index(2),
            left_c.set_z_index(3),
        )
        anims.append(Transform(left_c, right_c, path_arc=angle * DEGREES))

    self.play(*anims, run_time=2)
    self.wait()
"""

def __init__(
    self,
    mobject: Mobject | None,
    target_mobject: Mobject | None = None,
    path_func: Callable | None = None,
    path_arc: float = 0,
    path_arc_axis: np.ndarray = OUT,
    path_arc_centers: np.ndarray = None,
    replace_mobject_with_target_in_scene: bool = False,
    **kwargs,
) -> None:
    self.path_arc_axis: np.ndarray = path_arc_axis
    self.path_arc_centers: np.ndarray = path_arc_centers
    self.path_arc: float = path_arc

    # path_func is a property a few lines below so it doesn't need to be set in any case
    if path_func is not None:
        self.path_func: Callable = path_func
    elif self.path_arc_centers is not None:
        self.path_func = path_along_circles(
            path_arc,
            self.path_arc_centers,
            self.path_arc_axis,
        )

    self.replace_mobject_with_target_in_scene: bool = (
        replace_mobject_with_target_in_scene
    )
    self.target_mobject: Mobject = (
        target_mobject if target_mobject is not None else Mobject()
    )
    super().__init__(mobject, **kwargs)

@property
def path_arc(self) -> float:
    return self._path_arc

@path_arc.setter
def path_arc(self, path_arc: float) -> None:
    self._path_arc = path_arc
    self._path_func = path_along_arc(
        arc_angle=self._path_arc,
        axis=self.path_arc_axis,
    )

@property
def path_func(
    self,
) -> Callable[
    [Iterable[np.ndarray], Iterable[np.ndarray], float],
    Iterable[np.ndarray],
]:
    return self._path_func

@path_func.setter
def path_func(
    self,
    path_func: Callable[
        [Iterable[np.ndarray], Iterable[np.ndarray], float],
        Iterable[np.ndarray],
    ],
) -> None:
    if path_func is not None:
        self._path_func = path_func

def begin(self) -> None:
    # Use a copy of target_mobject for the align_data
    # call so that the actual target_mobject stays
    # preserved.
    self.target_mobject = self.create_target()
    self.target_copy = self.target_mobject.copy()
    # Note, this potentially changes the structure
    # of both mobject and target_mobject
    if config.renderer == RendererType.OPENGL:
        self.mobject.align_data_and_family(self.target_copy)
    else:
        self.mobject.align_data(self.target_copy)
    super().begin()

```

```

def create_target(self) -> Mobject:
    # Has no meaningful effect here, but may be useful
    # in subclasses
    return self.target_mobject

def clean_up_from_scene(self, scene: Scene) -> None:
    super().clean_up_from_scene(scene)
    if self.replace_mobject_with_target_in_scene:
        scene.replace(self.mobject, self.target_mobject)

def get_all_mobjects(self) -> Sequence[Mobject]:
    return [
        self.mobject,
        self.starting_mobject,
        self.target_mobject,
        self.target_copy,
    ]

def get_all_families_zipped(self) -> Iterable[tuple]: # more precise typing?
    mobs = [
        self.mobject,
        self.starting_mobject,
        self.target_copy,
    ]
    if config.renderer == RendererType.OPENGL:
        return zip(*[mob.get_family() for mob in mobs])
    return zip(*[mob.family_members_with_points() for mob in mobs])

def interpolate_submobject(
    self,
    submobject: Mobject,
    starting_submobject: Mobject,
    target_copy: Mobject,
    alpha: float,
) -> Transform:
    submobject.interpolate(starting_submobject, target_copy, alpha, self.path_func)
    return self

class ReplacementTransform(Transform):
    """Replaces and morphs a mobject into a target mobject.

    Parameters
    -----
    mobject
        The starting :class:`~.Mobject`.
    target_mobject
        The target :class:`~.Mobject`.
    kwargs
        Further keyword arguments that are passed to :class:`Transform`.

    Examples
    -----
    .. manim:: ReplacementTransformOrTransform
       :quality: low

    class ReplacementTransformOrTransform(Scene):
        def construct(self):
            # set up the numbers
            r_transform = VGroup(*[Integer(i) for i in range(1,4)])
            text_1 = Text("ReplacementTransform", color=RED)
            r_transform.add(text_1)

            transform = VGroup(*[Integer(i) for i in range(4,7)])
            text_2 = Text("Transform", color=BLUE)
            transform.add(text_2)

            ints = VGroup(r_transform, transform)
            texts = VGroup(text_1, text_2).scale(0.75)
            r_transform.arrange(direction=UP, buff=1)
            transform.arrange(direction=UP, buff=1)

            ints.arrange(buff=2)
            self.add(ints, texts)

            # The mobs replace each other and none are left behind
            self.play(ReplacementTransform(r_transform[0], r_transform[1]))
            self.play(ReplacementTransform(r_transform[1], r_transform[2]))

            # The mobs linger after the Transform()
            self.play(Transform(transform[0], transform[1]))
            self.play(Transform(transform[1], transform[2]))
            self.wait()

    """

    def __init__(self, mobject: Mobject, target_mobject: Mobject, **kwargs) -> None:
        super().__init__(
            mobject, target_mobject, replace_mobject_with_target_in_scene=True, **kwargs
        )

    class TransformFromCopy(Transform):
        """
        Performs a reversed Transform
        """

        def __init__(self, mobject: Mobject, target_mobject: Mobject, **kwargs) -> None:
            super().__init__(target_mobject, mobject, **kwargs)

        def interpolate(self, alpha: float) -> None:
            super().interpolate(1 - alpha)

```

```

class ClockwiseTransform(Transform):
    """Transforms the points of a mobject along a clockwise oriented arc.

See also
-----
:class:`.Transform`, :class:`.CounterclockwiseTransform`

Examples
-----

.. manim:: ClockwiseExample

class ClockwiseExample(Scene):
    def construct(self):
        dl, dr = Dot(), Dot()
        sl, sr = Square(), Square()

        VGroup(dl, sl).arrange(DOWN).shift(2*LEFT)
        VGroup(dr, sr).arrange(DOWN).shift(2*RIGHT)

        self.add(dl, dr)
        self.wait()
        self.play(
            ClockwiseTransform(dl, sl),
            Transform(dr, sr)
        )
        self.wait()

"""

def __init__(
    self,
    mobject: Mobject,
    target_mobject: Mobject,
    path_arc: float = -np.pi,
    **kwargs,
) -> None:
    super().__init__(mobject, target_mobject, path_arc=path_arc, **kwargs)

class CounterclockwiseTransform(Transform):
    """Transforms the points of a mobject along a counterclockwise oriented arc.

See also
-----
:class:`.Transform`, :class:`.ClockwiseTransform`

Examples
-----

.. manim:: CounterclockwiseTransform_vs_Transform

class CounterclockwiseTransform_vs_Transform(Scene):
    def construct(self):
        # set up the numbers
        c_transform = VGroup(DecimalNumber(number=3.141, num_decimal_places=3), DecimalNumber(number=1.618, num_decimal_places=3))
        text_1 = Text("CounterclockwiseTransform", color=RED)
        c_transform.add(text_1)

        transform = VGroup(DecimalNumber(number=1.618, num_decimal_places=3), DecimalNumber(number=3.141, num_decimal_places=3))
        text_2 = Text("Transform", color=BLUE)
        transform.add(text_2)

        ints = VGroup(c_transform, transform)
        texts = VGroup(text_1, text_2).scale(0.75)
        c_transform.arrange(direction=UP, buff=1)
        transform.arrange(direction=UP, buff=1)

        ints.arrange(buff=2)
        self.add(ints, texts)

        # The mobs move in clockwise direction for ClockwiseTransform()
        self.play(CounterclockwiseTransform(c_transform[0], c_transform[1]))

        # The mobs move straight up for Transform()
        self.play(Transform(transform[0], transform[1]))

"""

def __init__(
    self,
    mobject: Mobject,
    target_mobject: Mobject,
    path_arc: float = np.pi,
    **kwargs,
) -> None:
    super().__init__(mobject, target_mobject, path_arc=path_arc, **kwargs)

class MoveToTarget(Transform):
    """Transforms a mobject to the mobject stored in its ``target`` attribute.

After calling the :meth:`~.Mobject.generate_target` method, the :attr:`target` attribute of the mobject is populated with a copy of it. After modifying the attribute, playing the :class:`.MoveToTarget` animation transforms the original mobject into the modified one stored in the :attr:`target` attribute.

Examples
-----

.. manim:: MoveToTargetExample

class MoveToTargetExample(Scene):
    def construct(self):

```

```

    c = Circle()
    c.generate_target()
    c.target.set_fill(color=GREEN, opacity=0.5)
    c.target.shift(2*RIGHT + UP).scale(0.5)

    self.add(c)
    self.play(MoveToTarget(c))

"""

def __init__(self, mobject: Mobject, **kwargs) -> None:
    self.check_validity_of_input(mobject)
    super().__init__(mobject, mobject.target, **kwargs)

def check_validity_of_input(self, mobject: Mobject) -> None:
    if not hasattr(mobject, "target"):
        raise ValueError(
            "MoveToTarget called on mobject" "without attribute 'target'",
        )

class _MethodAnimation(MoveToTarget):
    def __init__(self, mobject, methods):
        self.methods = methods
        super().__init__(mobject)

    def finish(self) -> None:
        for method, method_args, method_kwargs in self.methods:
            method.__func__(self.mobject, *method_args, **method_kwargs)
        super().finish()

class ApplyMethod(Transform):
    """Animates a mobject by applying a method.

    Note that only the method needs to be passed to this animation,
    it is not required to pass the corresponding mobject. Furthermore,
    this animation class only works if the method returns the modified
    mobject.

    Parameters
    -----
    method
        The method that will be applied in the animation.
    args
        Any positional arguments to be passed when applying the method.
    kwargs
        Any keyword arguments passed to :class:`~.Transform`.

    """

    def __init__(
        self, method: Callable, *args, **kwargs
    ) -> None: # method typing (we want to specify Mobject method)? for args?
        self.check_validity_of_input(method)
        self.method = method
        self.method_args = args
        super().__init__(method.__self__, **kwargs)

    def check_validity_of_input(self, method: Callable) -> None:
        if not inspect.ismethod(method):
            raise ValueError(
                "Whoops, looks like you accidentally invoked "
                "the method you want to animate",
            )
        assert isinstance(method.__self__, (Mobject, OpenGLMobject))

    def create_target(self) -> Mobject:
        method = self.method
        # Make sure it's a list so that args.pop() works
        args = list(self.method_args)

        if len(args) > 0 and isinstance(args[-1], dict):
            method_kwargs = args.pop()
        else:
            method_kwargs = {}
        target = method.__self__.copy()
        method.__func__(target, *args, **method_kwargs)
        return target

class ApplyPointwiseFunction(ApplyMethod):
    """Animation that applies a pointwise function to a mobject.

    Examples
    -----
    .. manim:: WarpSquare
       :quality: low

       class WarpSquare(Scene):
           def construct(self):
               square = Square()
               self.play(
                   ApplyPointwiseFunction(
                       lambda point: complex_to_R3(np.exp(R3_to_complex(point))), square
                   )
               )
               self.wait()

    """

    def __init__(
        self,

```

```

function: types.MethodType,
mobject: Mobject,
run_time: float = DEFAULT_POINTWISE_FUNCTION_RUN_TIME,
**kwargs,
) -> None:
    super().__init__(mobject.apply_function, function, run_time=run_time, **kwargs)

class ApplyPointwiseFunctionToCenter(ApplyPointwiseFunction):
    def __init__(self, function: types.MethodType, mobject: Mobject, **kwargs) -> None:
        self.function = function
        super().__init__(mobject.move_to, **kwargs)

    def begin(self) -> None:
        self.method_args = [self.function(mobject.get_center())]
        super().begin()

class FadeToColor(ApplyMethod):
    """Animation that changes color of a mobject.

    Examples
    -------

    .. manim:: FadeToColorExample

        class FadeToColorExample(Scene):
            def construct(self):
                self.play(FadeToColor(Text("Hello World!"), color=RED))

    """

    def __init__(self, mobject: Mobject, color: str, **kwargs) -> None:
        super().__init__(mobject.set_color, color, **kwargs)

class ScaleInPlace(ApplyMethod):
    """Animation that scales a mobject by a certain factor.

    Examples
    -------

    .. manim:: ScaleInPlaceExample

        class ScaleInPlaceExample(Scene):
            def construct(self):
                self.play(ScaleInPlace(Text("Hello World!"), 2))

    """

    def __init__(self, mobject: Mobject, scale_factor: float, **kwargs) -> None:
        super().__init__(mobject.scale, scale_factor, **kwargs)

class ShrinkToCenter(ScaleInPlace):
    """Animation that makes a mobject shrink to center.

    Examples
    -------

    .. manim:: ShrinkToCenterExample

        class ShrinkToCenterExample(Scene):
            def construct(self):
                self.play(ShrinkToCenter(Text("Hello World!")))

    """

    def __init__(self, mobject: Mobject, **kwargs) -> None:
        super().__init__(mobject, 0, **kwargs)

class Restore(ApplyMethod):
    """Transforms a mobject to its last saved state.

    To save the state of a mobject, use the :meth:`~.Mobject.save_state` method.

    Examples
    -------

    .. manim:: RestoreExample

        class RestoreExample(Scene):
            def construct(self):
                s = Square()
                s.save_state()
                self.play(FadeIn(s))
                self.play(s.animate.set_color(PURPLE).set_opacity(0.5).shift(2*LEFT).scale(3))
                self.play(s.animate.shift(5*DOWN).rotate(PI/4))
                self.wait()
                self.play(Restore(s), run_time=2)

    """

    def __init__(self, mobject: Mobject, **kwargs) -> None:
        super().__init__(mobject.restore, **kwargs)

class ApplyFunction(Transform):
    def __init__(self, function: types.MethodType, mobject: Mobject, **kwargs) -> None:
        self.function = function
        super().__init__(mobject, **kwargs)

    def create_target(self) -> Any:
        target = self.function(mobject.copy())

```

```

if not isinstance(target, (Mobject, OpenGLMobject)):
    raise TypeError(
        "Functions passed to ApplyFunction must return object of type Mobject",
    )
return target

class ApplyMatrix(ApplyPointwiseFunction):
    """Applies a matrix transform to an mobject.

Parameters
-----
matrix
    The transformation matrix.
mobject
    The :class:`~.Mobject`.
about_point
    The origin point for the transform. Defaults to ``ORIGIN``.
kwargs
    Further keyword arguments that are passed to :class:`ApplyPointwiseFunction`.

Examples
-----
.. manim:: ApplyMatrixExample

    class ApplyMatrixExample(Scene):
        def construct(self):
            matrix = [[1, 1], [0, 2/3]]
            self.play(ApplyMatrix(matrix, Text("Hello World!")), ApplyMatrix(matrix, NumberPlane()))

    """

def __init__(self,
            matrix: np.ndarray,
            mobject: Mobject,
            about_point: np.ndarray = ORIGIN,
            **kwargs,
) -> None:
    matrix = self.initialize_matrix(matrix)

    def func(p):
        return np.dot(p - about_point, matrix.T) + about_point

    super().__init__(func, mobject, **kwargs)

def initialize_matrix(self, matrix: np.ndarray) -> np.ndarray:
    matrix = np.array(matrix)
    if matrix.shape == (2, 2):
        new_matrix = np.identity(3)
        new_matrix[:2, :2] = matrix
        matrix = new_matrix
    elif matrix.shape != (3, 3):
        raise ValueError("Matrix has bad dimensions")
    return matrix

class ApplyComplexFunction(ApplyMethod):
    def __init__(self, function: types.MethodType, mobject: Mobject, **kwargs) -> None:
        self.function = function
        method = mobject.apply_complex_function
        super().__init__(method, function, **kwargs)

    def __init_path_func(self) -> None:
        func1 = self.function(complex(1))
        self.path_arc = np.log(func1).imag
        super().__init_path_func()

    """

class CyclicReplace(Transform):
    """An animation moving mobjects cyclically.

In particular, this means: the first mobject takes the place
of the second mobject, the second one takes the place of
the third mobject, and so on. The last mobject takes the
place of the first one.

Parameters
-----
mobjects
    List of mobjects to be transformed.
path_arc
    The angle of the arc (in radians) that the mobjects will follow to reach
    their target.
kwargs
    Further keyword arguments that are passed to :class:`.Transform`.

Examples
-----
.. manim :: CyclicReplaceExample

    class CyclicReplaceExample(Scene):
        def construct(self):
            group = VGroup(Square(), Circle(), Triangle(), Star())
            group.arrange(RIGHT)
            self.add(group)

            for _ in range(4):
                self.play(CyclicReplace(*group))

    """

```

```

def __init__(self, *mobjects: Mobject, path_arc: float = 90 * DEGREES, **kwargs) -> None:
    self.group = Group(*mobjects)
    super().__init__(self.group, path_arc=path_arc, **kwargs)

def create_target(self) -> Group:
    target = self.group.copy()
    cycled_targets = [target[-1], *target[:-1]]
    for m1, m2 in zip(cycled_targets, self.group):
        m1.move_to(m2)
    return target

class Swap(CyclicReplace):
    pass # Renaming, more understandable for two entries

# TODO, this may be deprecated...worth reimplementing?
class TransformAnimations(Transform):
    def __init__(self, start_anim: Animation, end_anim: Animation, rate_func: Callable = squish_rate_func(smooth), **kwargs) -> None:
        self.start_anim = start_anim
        self.end_anim = end_anim
        if "run_time" in kwargs:
            self.run_time = kwargs.pop("run_time")
        else:
            self.run_time = max(start_anim.run_time, end_anim.run_time)
        for anim in start_anim, end_anim:
            anim.set_run_time(self.run_time)
        if (
            start_anim.starting_mobject is not None
            and end_anim.starting_mobject is not None
            and start_anim.starting_mobject.get_num_points() != end_anim.starting_mobject.get_num_points()
        ):
            start_anim.starting_mobject.align_data(end_anim.starting_mobject)
        for anim in start_anim, end_anim:
            if isinstance(anim, Transform) and anim.starting_mobject is not None:
                anim.starting_mobject.align_data(anim.target_mobject)

    super().__init__(start_anim.mobject, end_anim.mobject, rate_func=rate_func, **kwargs)
    # Rewire starting and ending mobjects
    start_anim.mobject = self.starting_mobject
    end_anim.mobject = self.target_mobject

    def interpolate(self, alpha: float) -> None:
        self.start_anim.interpolate(alpha)
        self.end_anim.interpolate(alpha)
        super().__interpolate(alpha)

class FadeTransform(Transform):
    """Fades one mobject into another.

    Parameters
    -----
    mobject
        The starting :class:`~.Mobject`.
    target_mobject
        The target :class:`~.Mobject`.
    stretch
        Controls whether the target :class:`~.Mobject` is stretched during the animation. Default: ``True``.
    dim_to_match
        If the target mobject is not stretched automatically, this allows to adjust the initial scale of the target :class:`~.Mobject` while it is shifted in. Setting this to 0, 1, and 2, respectively, matches the length of the target with the length of the starting :class:`~.Mobject` in x, y, and z direction, respectively.
    kwargs
        Further keyword arguments are passed to the parent class.

    Examples
    -----
    .. manim:: DifferentFadeTransforms

    class DifferentFadeTransforms(Scene):
        def construct(self):
            starts = [Rectangle(width=4, height=1) for _ in range(3)]
            VGroup(*starts).arrange(DOWN, buff=1).shift(3*LEFT)
            targets = [Circle(fill_opacity=1).scale(0.25) for _ in range(3)]
            VGroup(*targets).arrange(DOWN, buff=1).shift(3*RIGHT)

            self.play(*[FadeIn(s) for s in starts])
            self.play(
                FadeTransform(starts[0], targets[0], stretch=True),
                FadeTransform(starts[1], targets[1], stretch=False, dim_to_match=0),
                FadeTransform(starts[2], targets[2], stretch=False, dim_to_match=1)
            )
            self.play(*[FadeOut(mobj) for mobj in self.mobjects])

    """
    def __init__(self, mobject, target_mobject, stretch=True, dim_to_match=1, **kwargs):
        self.to_add_on_completion = target_mobject

```

```

self.stretch = stretch
self.dim_to_match = dim_to_match
mobject.save_state()
if config.renderer == RendererType.OPENGL:
    group = OpenGLGroup(mobject, target_mobject.copy())
else:
    group = Group(mobject, target_mobject.copy())
super().__init__(group, **kwargs)

def begin(self):
    """Initial setup for the animation.

    The mobject to which this animation is bound is a group consisting of
    both the starting and the ending mobject. At the start, the ending
    mobject replaces the starting mobject (and is completely faded). In the
    end, it is set to be the other way around.
    """
    self.ending_mobject = self.mobject.copy()
Animation.begin(self)
# Both 'start' and 'end' consists of the source and target mobjects.
# At the start, the target should be faded replacing the source,
# and at the end it should be the other way around.
start, end = self.starting_mobject, self.ending_mobject
for m0, m1 in ((start[1], start[0]), (end[0], end[1])):
    self.ghost_to(m0, m1)

def ghost_to(self, source, target):
    """Replaces the source by the target and sets the opacity to 0.

    If the provided target has no points, and thus a location of [0, 0, 0]
    the source will simply fade out where it currently is.
    """
    # mobject.replace() does not work if the target has no points.
    if target.get_num_points() or target.submobjects:
        source.replace(target, stretch=self.stretch, dim_to_match=self.dim_to_match)
    source.set_opacity(0)

def get_all_mobjects(self) -> Sequence[Mobject]:
    return [
        self.mobject,
        self.starting_mobject,
        self.ending_mobject,
    ]

def get_all_families_zipped(self):
    return Animation.get_all_families_zipped(self)

def clean_up_from_scene(self, scene):
    Animation.clean_up_from_scene(self, scene)
    scene.remove(self.mobject)
    self.mobject[0].restore()
    scene.add(self.to_add_on_completion)

class FadeTransformPieces(FadeTransform):
    """Fades submobjects of one mobject into submobjects of another one.

    See also
    -----
    :class:`~.FadeTransform`

    Examples
    -----
    .. manim:: FadeTransformSubmobjects

        class FadeTransformSubmobjects(Scene):
            def construct(self):
                src = VGroup(Square(), Circle().shift(LEFT + UP))
                src.shift(3*LEFT + 2*UP)
                src_copy = src.copy().shift(4*DOWN)

                target = VGroup(Circle(), Triangle().shift(RIGHT + DOWN))
                target.shift(3*RIGHT + 2*UP)
                target_copy = target.copy().shift(4*DOWN)

                self.play(FadeIn(src), FadeIn(src_copy))
                self.play(
                    FadeTransform(src, target),
                    FadeTransformPieces(src_copy, target_copy)
                )
                self.play(*[FadeOut(mobj) for mobj in self.mobjects])

    """

    def begin(self):
        self.mobject[0].align_submobjects(self.mobject[1])
        super().begin()

    def ghost_to(self, source, target):
        """Replaces the source submobjects by the target submobjects and sets
        the opacity to 0.
        """
        for sm0, sm1 in zip(source.get_family(), target.get_family()):
            super().ghost_to(sm0, sm1)

```

movement.py

```
"""Animations related to movement."""

from __future__ import annotations

__all__ = [
    "Homotopy",
    "SmoothedVectorizedHomotopy",
    "ComplexHomotopy",
    "PhaseFlow",
    "MoveAlongPath",
]
from typing import TYPE_CHECKING, Any, Callable
import numpy as np
from ..animation.animation import Animation
from ..utils.rate_functions import linear

if TYPE_CHECKING:
    from ..mobject.mobject import Mobject, VMobject

class Homotopy(Animation):
    """A Homotopy.

    This is an animation transforming the points of a mobject according
    to the specified transformation function. With the parameter :math:`t`
    moving from 0 to 1 throughout the animation and :math:`(x, y, z)`
    describing the coordinates of the point of a mobject,
    the function passed to the ``homotopy`` keyword argument should
    transform the tuple :math:`(x, y, z, t)` to :math:`(x', y', z')`,
    the coordinates the original point is transformed to at time :math:`t`.

    Parameters
    -----
    homotopy
        A function mapping :math:`(x, y, z, t)` to :math:`(x', y', z')`.
    mobject
        The mobject transformed under the given homotopy.
    run_time
        The run time of the animation.
    apply_function_kwds
        Keyword arguments propagated to :meth:`.Mobject.apply_function`.
    kwargs
        Further keyword arguments passed to the parent class.
    """

    def __init__(self,
                 homotopy: Callable[[float, float, float, float], tuple[float, float, float]],
                 mobject: Mobject,
                 run_time: float = 3,
                 apply_function_kwds: dict[str, Any] | None = None,
                 **kwargs,
    ) -> None:
        self.homotopy = homotopy
        self.apply_function_kwds = (
            apply_function_kwds if apply_function_kwds is not None else {}
        )
        super().__init__(mobject, run_time=run_time, **kwargs)

    def function_at_time_t(self, t: float) -> tuple[float, float, float]:
        return lambda p: self.homotopy(*p, t)

    def interpolate_submobject(
            self,
            submobject: Mobject,
            starting_submobject: Mobject,
            alpha: float,
    ) -> None:
        submobject.points = starting_submobject.points
        submobject.apply_function(
            self.function_at_time_t(alpha), **self.apply_function_kwds
        )

    class SmoothedVectorizedHomotopy(Homotopy):
        def interpolate_submobject(
                self,
                submobject: Mobject,
```

```

        starting_submobject: Mobject,
        alpha: float,
    ) -> None:
        super().interpolate_submobject(submobject, starting_submobject, alpha)
        submobject.make_smooth()

class ComplexHomotopy(Homotopy):
    def __init__(
        self, complex_homotopy: Callable[[complex], float], mobject: Mobject, **kwargs
    ) -> None:
        """
        Complex Homotopy a function Cx[0, 1] to C
        """
        def homotopy(
            x: float,
            y: float,
            z: float,
            t: float,
        ) -> tuple[float, float, float]:
            c = complex_homotopy(complex(x, y), t)
            return (c.real, c.imag, z)

        super().__init__(homotopy, mobject, **kwargs)

class PhaseFlow(Animation):
    def __init__(
        self,
        function: Callable[[np.ndarray], np.ndarray],
        mobject: Mobject,
        virtual_time: float = 1,
        suspend_mobject_updating: bool = False,
        rate_func: Callable[[float], float] = linear,
        **kwargs,
    ) -> None:
        self.virtual_time = virtual_time
        self.function = function
        super().__init__(
            mobject,
            suspend_mobject_updating=suspend_mobject_updating,
            rate_func=rate_func,
            **kwargs,
        )

    def interpolate_mobject(self, alpha: float) -> None:
        if hasattr(self, "last_alpha"):
            dt = self.virtual_time * (
                self.rate_func(alpha) - self.rate_func(self.last_alpha)
            )
            self.mobject.apply_function(lambda p: p + dt * self.function(p))
        self.last_alpha = alpha

class MoveAlongPath(Animation):
    """Make one mobject move along the path of another mobject.
    Example
    -----
    .. manim:: MoveAlongPathExample

    class MoveAlongPathExample(Scene):
        def construct(self):
            d1 = Dot().set_color(ORANGE)
            l1 = Line(LEFT, RIGHT)
            l2 = VMobject()
            self.add(d1, l1, l2)
            l2.add_updater(lambda x: x.become(Line(LEFT, d1.get_center()).set_color(ORANGE)))
            self.play(MoveAlongPath(d1, l1), rate_func=linear)
    """

    def __init__(
        self,
        mobject: Mobject,
        path: VMobject,
        suspend_mobject_updating: bool | None = False,
        **kwargs,
    ) -> None:
        self.path = path
        super().__init__(
            mobject, suspend_mobject_updating=suspend_mobject_updating, **kwargs
        )

    def interpolate_mobject(self, alpha: float) -> None:
        point = self.path.point_from_proportion(self.rate_func(alpha))
        self.mobject.move_to(point)

```

three_d_utils.py

```
"""Utility functions for three-dimensional mobjects."""
from __future__ import annotations

__all__ = [
    "get_3d_vmob_gradient_start_and_end_points",
    "get_3d_vmob_start_corner_index",
    "get_3d_vmob_end_corner_index",
    "get_3d_vmob_start_corner",
    "get_3d_vmob_end_corner",
    "get_3d_vmob_unit_normal",
    "get_3d_vmob_start_corner_unit_normal",
    "get_3d_vmob_end_corner_unit_normal",
]
import numpy as np

from manim.constants import ORIGIN, UP
from manim.utils.space_ops import get_unit_normal

def get_3d_vmob_gradient_start_and_end_points(vmob):
    return (
        get_3d_vmob_start_corner(vmob),
        get_3d_vmob_end_corner(vmob),
    )

def get_3d_vmob_start_corner_index(vmob):
    return 0

def get_3d_vmob_end_corner_index(vmob):
    return ((len(vmob.points) - 1) // 6) * 3

def get_3d_vmob_start_corner(vmob):
    if vmob.get_num_points() == 0:
        return np.array(ORIGIN)
    return vmob.points[get_3d_vmob_start_corner_index(vmob)]

def get_3d_vmob_end_corner(vmob):
    if vmob.get_num_points() == 0:
        return np.array(ORIGIN)
    return vmob.points[get_3d_vmob_end_corner_index(vmob)]

def get_3d_vmob_unit_normal(vmob, point_index):
    n_points = vmob.get_num_points()
    if len(vmob.get_anchors()) <= 2:
        return np.array(UP)
    i = point_index
    im3 = i - 3 if i > 2 else (n_points - 4)
    ip3 = i + 3 if i < (n_points - 3) else 3
    unit_normal = get_unit_normal(
        vmob.points[ip3] - vmob.points[i],
        vmob.points[im3] - vmob.points[i],
    )
    if np.linalg.norm(unit_normal) == 0:
        return np.array(UP)
    return unit_normal

def get_3d_vmob_start_corner_unit_normal(vmob):
    return get_3d_vmob_unit_normal(vmob, get_3d_vmob_start_corner_index(vmob))

def get_3d_vmob_end_corner_unit_normal(vmob):
    return get_3d_vmob_unit_normal(vmob, get_3d_vmob_end_corner_index(vmob))
```

svg_mobject.py

```
"""Mobjects generated from an SVG file."""

from __future__ import annotations

import os
from pathlib import Path
from xml.etree import ElementTree as ET

import numpy as np
import svgelements as se

from manim import config, logger

from ...constants import RIGHT
from ...utils.bezier import get_quadratic_approximation_of_cubic
from ...utils.images import get_full_vector_image_path
from ...utils.iterables import hash_obj
from ..geometry.arc import Circle
from ..geometry.line import Line
from ..geometry.polygram import Polygon, Rectangle, RoundedRectangle
from ..opengl.opengl_compatibility import ConvertToOpenGL
from ..types.vectorized_mobject import VMobject

__all__ = ["SVGMobject", "VMobjectFromSVGPath"]

SVG_HASH_TO_MOB_MAP: dict[int, VMobject] = {}

def _convert_point_to_3d(x: float, y: float) -> np.ndarray:
    return np.array([x, y, 0.0])

class SVGMobject(VMobject, metaclass=ConvertToOpenGL):
    """A vectorized mobject created from importing an SVG file.

    Parameters
    -----
    file_name
        The path to the SVG file.
    should_center
        Whether or not the mobject should be centered after
        being imported.
    height
        The target height of the mobject, set to 2 Manim units by default.
        If the height and width are both set to ``None``, the mobject
        is imported without being scaled.
    width
        The target width of the mobject, set to ``None`` by default. If
        the height and the width are both set to ``None``, the mobject
        is imported without being scaled.
    color
        The color (both fill and stroke color) of the mobject. If
        ``None`` (the default), the colors set in the SVG file
        are used.
    opacity
        The opacity (both fill and stroke opacity) of the mobject.
        If ``None`` (the default), the opacity set in the SVG file
        is used.
    fill_color
        The fill color of the mobject. If ``None`` (the default),
        the fill colors set in the SVG file are used.
    fill_opacity
        The fill opacity of the mobject. If ``None`` (the default),
        the fill opacities set in the SVG file are used.
    stroke_color
        The stroke color of the mobject. If ``None`` (the default),
        the stroke colors set in the SVG file are used.
    stroke_opacity
        The stroke opacity of the mobject. If ``None`` (the default),
        the stroke opacities set in the SVG file are used.
    stroke_width
        The stroke width of the mobject. If ``None`` (the default),
        the stroke width values set in the SVG file are used.
    svg_default
        A dictionary in which fallback values for unspecified
        properties of elements in the SVG file are defined. If
        ``None`` (the default), ``color``, ``opacity``, ``fill_color``,
        ``fill_opacity``, ``stroke_color``, and ``stroke_opacity``
        are set to ``None``, and ``stroke_width`` is set to 0.
    path_string_config
```

```

A dictionary with keyword arguments passed to
:func:`.VMobjectFromSVGPath` used for importing path elements.
If ``None`` (the default), no additional arguments are passed.

use_svg_cache
    If ``True`` (default), the svg inputs (e.g. file_name, settings)
    will be used as a key and a copy of the created mobject will
    be saved using that key to be quickly retrieved if the same
    inputs need be processed later. For large SVGs which are used
    only once, this can be omitted to improve performance.

kwargs
    Further arguments passed to the parent class.

"""

def __init__(
    self,
    file_name: str | os.PathLike | None = None,
    should_center: bool = True,
    height: float | None = 2,
    width: float | None = None,
    color: str | None = None,
    opacity: float | None = None,
    fill_color: str | None = None,
    fill_opacity: float | None = None,
    stroke_color: str | None = None,
    stroke_opacity: float | None = None,
    stroke_width: float | None = None,
    svg_default: dict | None = None,
    path_string_config: dict | None = None,
    use_svg_cache: bool = True,
    **kwargs,
):
    super().__init__(color=None, stroke_color=None, fill_color=None, **kwargs)

    # process keyword arguments
    self.file_name = Path(file_name) if file_name is not None else None

    self.should_center = should_center
    self.svg_height = height
    self.svg_width = width
    self.color = color
    self.opacity = opacity
    self.fill_color = fill_color
    self.fill_opacity = fill_opacity
    self.stroke_color = stroke_color
    self.stroke_opacity = stroke_opacity
    self.stroke_width = stroke_width

    if svg_default is None:
        svg_default = {
            "color": None,
            "opacity": None,
            "fill_color": None,
            "fill_opacity": None,
            "stroke_width": 0,
            "stroke_color": None,
            "stroke_opacity": None,
        }
    self.svg_default = svg_default

    if path_string_config is None:
        path_string_config = {}
    self.path_string_config = path_string_config

    self.init_svg_mobject(use_svg_cache=use_svg_cache)

    self.set_style(
        fill_color=fill_color,
        fill_opacity=fill_opacity,
        stroke_color=stroke_color,
        stroke_opacity=stroke_opacity,
        stroke_width=stroke_width,
    )
    self.move_into_position()

def init_svg_mobject(self, use_svg_cache: bool) -> None:
    """Checks whether the SVG has already been imported and
    generates it if not.

    See also
    -----
    :meth:`.SVGMobject.generate_mobject`
    """
    if use_svg_cache:
        hash_val = hash_obj(self.hash_seed)
        if hash_val in SVG_HASH_TO_MOB_MAP:
            mob = SVG_HASH_TO_MOB_MAP[hash_val].copy()
            self.add(*mob)

```

```

        return

    self.generate_mobject()
    if use_svg_cache:
        SVG_HASH_TO_MOB_MAP[hash_val] = self.copy()

@property
def hash_seed(self) -> tuple:
    """A unique hash representing the result of the generated
    mobject points.

    Used as keys in the ``SVG_HASH_TO_MOB_MAP`` caching dictionary.
    """
    return (
        self.__class__.__name__,
        self.svg_default,
        self.path_string_config,
        self.file_name,
        config.renderer,
    )

def generate_mobject(self) -> None:
    """Parse the SVG and translate its elements to subobjects."""
    file_path = self.get_file_path()
    element_tree = ET.parse(file_path)
    new_tree = self.modify_xml_tree(element_tree)
    # Create a temporary svg file to dump modified svg to be parsed
    modified_file_path = file_path.with_name(f"{file_path.stem}_{file_path.suffix}")
    new_tree.write(modified_file_path)

    svg = se.SVG.parse(modified_file_path)
    modified_file_path.unlink()

    mobjects = self.get_mobjects_from(svg)
    self.add(*mobjects)
    self.flip(RIGHT) # Flip y

def get_file_path(self) -> Path:
    """Search for an existing file based on the specified file name."""
    if self.file_name is None:
        raise ValueError("Must specify file for SVGObject")
    return get_full_vector_image_path(self.file_name)

def modify_xml_tree(self, element_tree: ET.ElementTree) -> ET.ElementTree:
    """Modifies the SVG element tree to include default
    style information.

    Parameters
    -----
    element_tree
        The parsed element tree from the SVG file.
    """
    config_style_dict = self.generate_config_style_dict()
    style_keys = (
        "fill",
        "fill-opacity",
        "stroke",
        "stroke-opacity",
        "stroke-width",
        "style",
    )
    root = element_tree.getroot()
    root_style_dict = {k: v for k, v in root.attrib.items() if k in style_keys}

    new_root = ET.Element("svg", {})
    config_style_node = ET.SubElement(new_root, "g", config_style_dict)
    root_style_node = ET.SubElement(config_style_node, "g", root_style_dict)
    root_style_node.extend(root)
    return ET.ElementTree(new_root)

def generate_config_style_dict(self) -> dict[str, str]:
    """Generate a dictionary holding the default style information."""
    keys_converting_dict = {
        "fill": ("color", "fill_color"),
        "fill-opacity": ("opacity", "fill_opacity"),
        "stroke": ("color", "stroke_color"),
        "stroke-opacity": ("opacity", "stroke_opacity"),
        "stroke-width": ("stroke_width",),
    }
    svg_default_dict = self.svg_default
    result = {}
    for svg_key, style_keys in keys_converting_dict.items():
        for style_key in style_keys:
            if svg_default_dict[style_key] is None:
                continue
            result[svg_key] = str(svg_default_dict[style_key])
    return result

```

```

def get_mobjects_from(self, svg: se.SVG) -> list[VMobject]:
    """Convert the elements of the SVG to a list of mobjects.

    Parameters
    -----
    svg
        The parsed SVG file.
    """
    result = []
    for shape in svg.elements():
        if isinstance(shape, se.Group):
            continue
        elif isinstance(shape, se.Path):
            mob = self.path_to_mobject(shape)
        elif isinstance(shape, se.SimpleLine):
            mob = self.line_to_mobject(shape)
        elif isinstance(shape, se.Rect):
            mob = self.rect_to_mobject(shape)
        elif isinstance(shape, (se.Circle, se.Ellipse)):
            mob = self.ellipse_to_mobject(shape)
        elif isinstance(shape, se.Polygon):
            mob = self.polygon_to_mobject(shape)
        elif isinstance(shape, se.Polyline):
            mob = self.polyline_to_mobject(shape)
        elif isinstance(shape, se.Text):
            mob = self.text_to_mobject(shape)
        elif isinstance(shape, se.Use) or type(shape) == se.SVGElement:
            continue
        else:
            logger.warning(f"Unsupported element type: {type(shape)}")
            continue
        if mob is None or not mob.has_points():
            continue
        self.apply_style_to_mobject(mob, shape)
        if isinstance(shape, se.Transformable) and shape.apply:
            self.handle_transform(mob, shape.transform)
        result.append(mob)
    return result

@staticmethod
def handle_transform(mob: VMobject, matrix: se.Matrix) -> VMobject:
    """Apply SVG transformations to the converted mobject.

    Parameters
    -----
    mob
        The converted mobject.
    matrix
        The transformation matrix determined from the SVG
        transformation.
    """
    mat = np.array([[matrix.a, matrix.c], [matrix.b, matrix.d]])
    vec = np.array([matrix.e, matrix.f, 0.0])
    mob.apply_matrix(mat)
    mob.shift(vec)
    return mob

@staticmethod
def apply_style_to_mobject(mob: VMobject, shape: se.GraphicObject) -> VMobject:
    """Apply SVG style information to the converted mobject.

    Parameters
    -----
    mob
        The converted mobject.
    shape
        The parsed SVG element.
    """
    mob.set_style(
        stroke_width=shape.stroke_width,
        stroke_color=shape.stroke.hexrgb,
        stroke_opacity=shape.stroke.opacity,
        fill_color=shape.fill.hexrgb,
        fill_opacity=shape.fill.opacity,
    )
    return mob

def path_to_mobject(self, path: se.Path) -> VMobjectFromSVGPath:
    """Convert a path element to a vectorized mobject.

    Parameters
    -----
    path
        The parsed SVG path.
    """
    return VMobjectFromSVGPath(path, **self.path_string_config)

```

```

@staticmethod
def line_to_mobject(line: se.Line) -> Line:
    """Convert a line element to a vectorized mobject.

    Parameters
    -----
    line
        The parsed SVG line.
    """
    return Line(
        start=_convert_point_to_3d(line.x1, line.y1),
        end=_convert_point_to_3d(line.x2, line.y2),
    )

@staticmethod
def rect_to_mobject(rect: se.Rect) -> Rectangle:
    """Convert a rectangle element to a vectorized mobject.

    Parameters
    -----
    rect
        The parsed SVG rectangle.
    """
    if rect.rx == 0 or rect.ry == 0:
        mob = Rectangle(
            width=rect.width,
            height=rect.height,
        )
    else:
        mob = RoundedRectangle(
            width=rect.width,
            height=rect.height * rect.rx / rect.ry,
            corner_radius=rect.rx,
        )
        mob.stretch_to_fit_height(rect.height)
    mob.shift(_convert_point_to_3d(rect.x + rect.width / 2, rect.y + rect.height / 2))
    return mob

@staticmethod
def ellipse_to_mobject(ellipse: se.Ellipse | se.Circle) -> Circle:
    """Convert an ellipse or circle element to a vectorized mobject.

    Parameters
    -----
    ellipse
        The parsed SVG ellipse or circle.
    """
    mob = Circle(radius=ellipse.rx)
    if ellipse.rx != ellipse.ry:
        mob.stretch_to_fit_height(2 * ellipse.ry)
    mob.shift(_convert_point_to_3d(ellipse.cx, ellipse.cy))
    return mob

@staticmethod
def polygon_to_mobject(polygon: se.Polygon) -> Polygon:
    """Convert a polygon element to a vectorized mobject.

    Parameters
    -----
    polygon
        The parsed SVG polygon.
    """
    points = [_convert_point_to_3d(*point) for point in polygon]
    return Polygon(*points)

def polyline_to_mobject(self, polyline: se.Polyline) -> VMobject:
    """Convert a polyline element to a vectorized mobject.

    Parameters
    -----
    polyline
        The parsed SVG polyline.
    """
    points = [_convert_point_to_3d(*point) for point in polyline]
    vmobject_class = self.get_mobject_type_class()
    return vmobject_class().set_points_as_corners(points)

@staticmethod
def text_to_mobject(text: se.Text):
    """Convert a text element to a vectorized mobject.

    .. warning::

        Not yet implemented.
    """

```

```

Parameters
-----
text
    The parsed SVG text.
"""
logger.warning(f"Unsupported element type: {type(text)}")
return

def move_into_position(self) -> None:
    """Scale and move the generated mobject into position."""
    if self.should_center:
        self.center()
    if self.svg_height is not None:
        self.set_height(self.svg_height)
    if self.svg_width is not None:
        self.set_width(self.svg_width)

class VMobjectFromSVGPath(VMobject, metaclass=ConvertToOpenGL):
    """A vectorized mobject representing an SVG path.

.. note::

    The ``long_lines``, ``should_subdivide_sharp_curves``,
    and ``should_remove_null_curves`` keyword arguments are
    only respected with the OpenGL renderer.

Parameters
-----
path_obj
    A parsed SVG path object.
long_lines
    Whether or not straight lines in the vectorized mobject
    are drawn in one or two segments.
should_subdivide_sharp_curves
    Whether or not to subdivide subcurves further in case
    two segments meet at an angle that is sharper than a
    given threshold.
should_remove_null_curves
    Whether or not to remove subcurves of length 0.
kwargs
    Further keyword arguments are passed to the parent
    class.
"""

def __init__(
    self,
    path_obj: se.Path,
    long_lines: bool = False,
    should_subdivide_sharp_curves: bool = False,
    should_remove_null_curves: bool = False,
    **kwargs,
):
    # Get rid of arcs
    path_obj.approximate_arcs_with_quads()
    self.path_obj = path_obj

    self.long_lines = long_lines
    self.should_subdivide_sharp_curves = should_subdivide_sharp_curves
    self.should_remove_null_curves = should_remove_null_curves

    super().__init__(**kwargs)

def init_points(self) -> None:
    # TODO: cache mobject in a re-importable way

    self.handle_commands()

    if config.renderer == "opengl":
        if self.should_subdivide_sharp_curves:
            # For a healthy triangulation later
            self.subdivide_sharp_curves()
        if self.should_remove_null_curves:
            # Get rid of any null curves
            self.set_points(self.get_points_without_null_curves())

generate_points = init_points

def handle_commands(self) -> None:
    all_points: list[np.ndarray] = []
    last_move = None
    curve_start = None

    # These lambdas behave the same as similar functions in
    # vectorized_mobject, except they add to a list of points instead
    # of updating this Mobject's numpy array of points. This way,

```

```

# we don't observe O(n^2) behavior for complex paths due to
# numpy's need to re-allocate memory on every append.
def move_pen(pt):
    nonlocal last_move, curve_start
    last_move = pt
    if curve_start is None:
        curve_start = last_move

if self.n_points_per_curve == 4:

    def add_cubic(start, cp1, cp2, end):
        nonlocal all_points
        assert len(all_points) % 4 == 0, len(all_points)
        all_points += [start, cp1, cp2, end]
        move_pen(end)

    def add_quad(start, cp, end):
        add_cubic(start, (start + cp + cp) / 3, (cp + cp + end) / 3, end)
        move_pen(end)

    def add_line(start, end):
        add_cubic(
            start, (start + start + end) / 3, (start + end + end) / 3, end
        )
        move_pen(end)

else:

    def add_cubic(start, cp1, cp2, end):
        nonlocal all_points
        assert len(all_points) % 3 == 0, len(all_points)
        two_quads = get_quadratic_approximation_of_cubic(
            start,
            cp1,
            cp2,
            end,
        )
        all_points += two_quads[:3].tolist()
        all_points += two_quads[3:].tolist()
        move_pen(end)

    def add_quad(start, cp, end):
        nonlocal all_points
        assert len(all_points) % 3 == 0, len(all_points)
        all_points += [start, cp, end]
        move_pen(end)

    def add_line(start, end):
        add_quad(start, (start + end) / 2, end)
        move_pen(end)

for segment in self.path_obj:
    segment_class = segment.__class__
    if segment_class == se.Move:
        move_pen(_convert_point_to_3d(*segment.end))
    elif segment_class == se.Line:
        add_line(last_move, _convert_point_to_3d(*segment.end))
    elif segment_class == se.QuadraticBezier:
        add_quad(
            last_move,
            _convert_point_to_3d(*segment.control),
            _convert_point_to_3d(*segment.end),
        )
    elif segment_class == se.CubicBezier:
        add_cubic(
            last_move,
            _convert_point_to_3d(*segment.control1),
            _convert_point_to_3d(*segment.control2),
            _convert_point_to_3d(*segment.end),
        )
    elif segment_class == se.Close:
        # If the SVG path naturally ends at the beginning of the curve,
        # we do *not* need to draw a closing line. To account for floating
        # point precision, we use a small value to compare the two points.
        if abs(np.linalg.norm(last_move - curve_start)) > 0.0001:
            add_line(last_move, curve_start)
            curve_start = None
    else:
        raise AssertionError(f"Not implemented: {segment_class}")

self.points = np.array(all_points, ndmin=2, dtype="float64")
# If we have no points, make sure the array is shaped properly
# (0 rows tall by 3 columns wide) so future operations can
# add or remove points correctly.
if len(all_points) == 0:
    self.points = np.reshape(self.points, (0, 3))

```

render_options.py

```

from __future__ import annotations

import re

import click
from cloup import option, option_group

from manim.constants import QUALITIES, RendererType

from ... import logger

def validate_scene_range(ctx, param, value):
    try:
        start = int(value)
        return (start,)
    except Exception:
        pass

    if value:
        try:
            start, end = map(int, re.split(r";|-", value))
            return start, end
        except Exception:
            logger.error("Couldn't determine a range for -n option.")
            exit()

def validate_resolution(ctx, param, value):
    if value:
        try:
            start, end = map(int, re.split(r";|-", value))
            return (start, end)
        except Exception:
            logger.error("Resolution option is invalid.")
            exit()

render_options = option_group(
    "Render Options",
    option(
        "-n",
        "--from_animation_number",
        callback=validate_scene_range,
        help="Start rendering from n_0 until n_1. If n_1 is left unspecified, " +
        "renders all scenes after n_0.",
        default=None,
    ),
    option(
        "-a",
        "--write_all",
        is_flag=True,
        help="Render all scenes in the input file.",
        default=None,
    ),
    option(
        "--format",
        type=click.Choice(["png", "gif", "mp4", "webm", "mov"], case_sensitive=False),
        default=None,
    ),
    option("-s", "--save_last_frame", is_flag=True, default=None),
    option(
        "-q",
        "--quality",
        default=None,
        type=click.Choice(
            list(reversed([q["flag"] for q in QUALITIES.values() if q["flag"]])), # type: ignore
            case_sensitive=False,
        ),
        help="Render quality at the follow resolution framerates, respectively: " +
        "+ ".join(
            reversed([
                f'{q["pixel_width"]}x{q["pixel_height"]} {q["frame_rate"]}FPS'
                for q in QUALITIES.values()
                if q["flag"]
            ])
        ),
    ),
    option(

```

```

"-r",
"--resolution",
callback=validate_resolution,
default=None,
help='Resolution in "W,H" for when 16:9 aspect ratio isn\'t possible.',
),
option(
    "--fps",
    "--frame_rate",
    "frame_rate",
    type=float,
    default=None,
    help="Render at this frame rate.",
),
option(
    "--renderer",
    type=click.Choice(
        [renderer_type.value for renderer_type in RendererType],
        case_sensitive=False,
    ),
    help="Select a renderer for your Scene.",
    default="cairo",
),
option(
    "-g",
    "--save_pngs",
    is_flag=True,
    default=None,
    help="Save each frame as png (Deprecated).",
),
option(
    "-i",
    "--save_as_gif",
    default=None,
    is_flag=True,
    help="Save as a gif (Deprecated).",
),
option(
    "--save_sections",
    default=None,
    is_flag=True,
    help="Save section videos in addition to movie file.",
),
option(
    "-s",
    "--save_last_frame",
    default=None,
    is_flag=True,
    help="Save last frame as png (Deprecated).",
),
option(
    "-t",
    "--transparent",
    is_flag=True,
    help="Render scenes with alpha channel.",
),
option(
    "--use_projection_fill_shaders",
    is_flag=True,
    help="Use shaders for OpenGLObject fill which are compatible with transformation matrices.",
    default=None,
),
option(
    "--use_projection_stroke_shaders",
    is_flag=True,
    help="Use shaders for OpenGLObject stroke which are compatible with transformation matrices.",
    default=None,
),
)
)

```

manim_directive.py

```
r"""
A directive for including Manim videos in a Sphinx document
=====
When rendering the HTML documentation, the ``.. manim::`` directive
implemented here allows to include rendered videos.
```

```
Its basic usage that allows processing **inline content**
looks as follows::
```

```
.. manim:: MyScene
```

```
class MyScene(Scene):
    def construct(self):
        ...
```

```
It is required to pass the name of the class representing the
scene to be rendered to the directive.
```

```
As a second application, the directive can also be used to
render scenes that are defined within doctests, for example::
```

```
.. manim:: DirectiveDoctestExample
:ref_classes: Dot

>>> from manim import Create, Dot, RED, Scene
>>> dot = Dot(color=RED)
>>> dot.color
<Color #fc6255>
>>> class DirectiveDoctestExample(Scene):
...     def construct(self):
...         self.play(Create(dot))
```

```
Options
-----

```

```
Options can be passed as follows::
```

```
.. manim:: <Class name>
:<option name>: <value>
```

```
The following configuration options are supported by the
directive:
```

```
hide_source
If this flag is present without argument,
the source code is not displayed above the rendered video.
```

```
no_autoplay
If this flag is present without argument,
the video will not autoplay.
```

```
quality : {'low', 'medium', 'high', 'fourk'}
Controls render quality of the video, in analogy to
the corresponding command line flags.
```

```
save_as_gif
If this flag is present without argument,
the scene is rendered as a gif.
```

```
save_last_frame
If this flag is present without argument,
an image representing the last frame of the scene will
be rendered and displayed, instead of a video.
```

```
ref_classes
A list of classes, separated by spaces, that is
rendered in a reference block after the source code.
```

```
ref_functions
A list of functions, separated by spaces,
that is rendered in a reference block after the source code.
```

```
ref_methods
A list of methods, separated by spaces,
that is rendered in a reference block after the source code.
```

```
"""

```

```
from __future__ import annotations
```

```
import csv
import itertools as it
import os
import re
```

```

import shutil
import sys
from pathlib import Path
from timeit import timeit

import jinja2
from docutils import nodes
from docutils.parsers.rst import Directive, directives # type: ignore
from docutils.statemachine import StringList

from manim import QUALITIES

classnamesdict = {}

class SkipManimNode(nodes.Admonition, nodes.Element):
    """Auxiliary node class that is used when the ``skip-manim`` tag is present
    or ``.pot`` files are being built.

    Skips rendering the manim directive and outputs a placeholder instead.
    """

    pass

def visit(self, node, name=""):
    self.visit_admonition(node, name)
    if not isinstance(node[0], nodes.title):
        node.insert(0, nodes.title("skip-manim", "Example Placeholder"))

def depart(self, node):
    self.depart_admonition(node)

def process_name_list(option_input: str, reference_type: str) -> list[str]:
    """Reformats a string of space separated class names
    as a list of strings containing valid Sphinx references.

    Tests
    -----
    ::

        >>> process_name_list("Tex TexTemplate", "class")
        [':class:`~.Tex`', ':class:`~.TexTemplate`']
        >>> process_name_list("Scene.play Mobject.rotate", "func")
        [:func:`~.Scene.play`, :func:`~.Mobject.rotate`]
    """
    return [f":{reference_type}:{`~.{name}`}" for name in option_input.split()]

class ManimDirective(Directive):
    """The manim directive, rendering videos while building
    the documentation.

    See the module docstring for documentation.
    """

    has_content = True
    required_arguments = 1
    optional_arguments = 0
    option_spec = {
        "hide_source": bool,
        "no_autoplay": bool,
        "quality": lambda arg: directives.choice(
            arg,
            ("low", "medium", "high", "fourk"),
        ),
        "save_as_gif": bool,
        "save_last_frame": bool,
        "ref_modules": lambda arg: process_name_list(arg, "mod"),
        "ref_classes": lambda arg: process_name_list(arg, "class"),
        "ref_functions": lambda arg: process_name_list(arg, "func"),
        "ref_methods": lambda arg: process_name_list(arg, "meth"),
    }
    final_argument_whitespace = True

    def run(self):
        # Rendering is skipped if the tag skip-manim is present,
        # or if we are making the pot-files
        should_skip = (
            "skip-manim" in self.state.document.settings.env.app.builder.tags.tags
            or self.state.document.settings.env.app.builder.name == "gettext"
        )
        if should_skip:
            node = SkipManimNode()
            self.state.nested_parse(
                StringList(
                    [
                        f"Placeholder block for ``{self.arguments[0]}``.",
                        ""
                    ],
                    None
                ),
                self.state.document.settings.env.app.builder.current_docname
            )
            return [node]
        else:
            return super().run()

```

```

        "... code-block:: python",
        "",
    ]
+ ["    " + line for line in self.content]
),
self.content_offset,
node,
)
return [node]

from manim import config, tempconfig

global classnamedict

classname = self.arguments[0]
if classname not in classnamedict:
    classnamedict[classname] = 1
else:
    classnamedict[classname] += 1

hide_source = "hide_source" in self.options
no_autoplay = "no_autoplay" in self.options
save_as_gif = "save_as_gif" in self.options
save_last_frame = "save_last_frame" in self.options
assert not (save_as_gif and save_last_frame)

ref_content = (
    self.options.get("ref_modules", [])
+ self.options.get("ref_classes", [])
+ self.options.get("ref_functions", [])
+ self.options.get("ref_methods", [])
)
if ref_content:
    ref_block = "References: " + "\n".join(ref_content)
else:
    ref_block = ""

if "quality" in self.options:
    quality = f'{self.options["quality"]}_quality'
else:
    quality = "example_quality"
frame_rate = QUALITIES[quality]["frame_rate"]
pixel_height = QUALITIES[quality]["pixel_height"]
pixel_width = QUALITIES[quality]["pixel_width"]

state_machine = self.state_machine
document = state_machine.document

source_file_name = Path(document.attributes["source"])
source_rel_name = source_file_name.relative_to(setup.confdir)
source_rel_dir = source_rel_name.parents[0]
dest_dir = Path(setup.app.builder.outdir, source_rel_dir).absolute()
if not dest_dir.exists():
    dest_dir.mkdir(parents=True, exist_ok=True)

source_block = [
    "... code-block:: python",
    "",
    "    from manim import *\n",
    *("    " + line for line in self.content),
]
source_block = "\n".join(source_block)

config.media_dir = (Path(setup.confdir) / "media").absolute()
config.images_dir = "{media_dir}/images"
config.video_dir = "{media_dir}/videos/{quality}"
output_file = f"{classname}-{classnamesdict[classname]}"
config.assets_dir = Path("_static")
config.progress_bar = "none"
config.verbosity = "WARNING"

example_config = {
    "frame_rate": frame_rate,
    "no_autoplay": no_autoplay,
    "pixel_height": pixel_height,
    "pixel_width": pixel_width,
    "save_last_frame": save_last_frame,
    "write_to_movie": not save_last_frame,
    "output_file": output_file,
}
if save_last_frame:
    example_config["format"] = None
if save_as_gif:
    example_config["format"] = "gif"

user_code = self.content
if user_code[0].startswith(">>> "): # check whether block comes from doctest
    user_code = [
        line[4:] for line in user_code if line.startswith((">>> ", "... "))
    ]

```

```

code = [
    "from manim import *",
    "*user_code",
    f"\"{clsname}().render()\"",
]
try:
    with tempconfig(example_config):
        run_time = timeit(lambda: exec("\n".join(code), globals()), number=1)
        video_dir = config.get_dir("video_dir")
        images_dir = config.get_dir("images_dir")
except Exception as e:
    raise RuntimeError(f"Error while rendering example {clsname}") from e

_write_rendering_stats(
    clsname,
    run_time,
    self.state.document.settings.env.docname,
)

# copy video file to output directory
if not (save_as_gif or save_last_frame):
    filename = f"{output_file}.mp4"
    filesrc = video_dir / filename
    destfile = Path(dest_dir, filename)
    shutil.copyfile(filesrc, destfile)
elif save_as_gif:
    filename = f"{output_file}.gif"
    filesrc = video_dir / filename
elif save_last_frame:
    filename = f"{output_file}.png"
    filesrc = images_dir / filename
else:
    raise ValueError("Invalid combination of render flags received.")
rendered_template = jinja2.Template(TEMPLATE).render(
    clsname=clsname,
    clsname_lowercase=clsname.lower(),
    hide_source=hide_source,
    filesrc_rel=Path(filesrc).relative_to(setup.confdir).as_posix(),
    no_autoplay=no_autoplay,
    output_file=output_file,
    save_last_frame=save_last_frame,
    save_as_gif=save_as_gif,
    source_block=source_block,
    ref_block=ref_block,
)
state_machine.insert_input(
    rendered_template.split("\n"),
    source=document.attributes["source"],
)
return []

rendering_times_file_path = Path("../rendering_times.csv")

def _write_rendering_stats(scene_name, run_time, file_name):
    with rendering_times_file_path.open("a") as file:
        csv.writer(file).writerow([
            re.sub(r"^(reference\/)|(manim\.)", "", file_name),
            scene_name,
            "%.3f" % run_time,
        ],
    )

def _log_rendering_times(*args):
    if rendering_times_file_path.exists():
        with rendering_times_file_path.open() as file:
            data = list(csv.reader(file))
    if len(data) == 0:
        sys.exit()

    print("\nRendering Summary\n-----\n")

    max_file_length = max(len(row[0]) for row in data)
    for key, group in it.groupby(data, key=lambda row: row[0]):
        key = key.ljust(max_file_length + 1, ".")
        group = list(group)
        if len(group) == 1:
            row = group[0]
            print(f"{key}{row[2].rjust(7, '.')}{row[1]}")
            continue
        time_sum = sum(float(row[2]) for row in group)
        print(
            f"{key}{f'{time_sum:.3f}'.rjust(7, '.')}{len(group)} EXAMPLES",
        )
        for row in group:

```

```

        print(f"{' '* (max_file_length)} {row[2].rjust(7)}s {row[1]}'")
    print("")

def _delete_rendering_times(*args):
    if rendering_times_file_path.exists():
        rendering_times_file_path.unlink()

def setup(app):
    app.add_node(SkipManimNode, html=(visit, depart))

    setup.app = app
    setup.config = app.config
    setup.confdir = app.confdir

    app.add_directive("manim", ManimDirective)

    app.connect("builder-initiated", _delete_rendering_times)
    app.connect("build-finished", _log_rendering_times)

    metadata = {"parallel_read_safe": False, "parallel_write_safe": True}
    return metadata

TEMPLATE = r"""
{%
    if not hide_source %}
    .. raw:: html

        <div id="{{ clsname_lowercase }}" class="admonition admonition-manim-example">
            <p class="admonition-title">Example: {{ clsname }} <a class="headerlink" href="#{{ clsname_lowercase }}">Â¶</a></p>
        {% endif %}

    {% if not (save_as_gif or save_last_frame) %}
    .. raw:: html

        <video
            class="manim-video"
            controls
            loop
            {{ '' if no_autoplay else 'autoplay' }}
            src="./{{ output_file }}.mp4">
        </video>

    {% elif save_as_gif %}
    .. image:: /{{ filesrc_rel }}
        :align: center

    {% elif save_last_frame %}
    .. image:: /{{ filesrc_rel }}
        :align: center

    {% endif %}
    {% if not hide_source %}
    {{ source_block }}

    {{ ref_block }}

    .. raw:: html

        </div>
    {% endif %}
    """

```

opengl_mobject.py

```
from __future__ import annotations

import copy
import itertools as it
import random
import sys
from functools import partialmethod, wraps
from math import ceil
from typing import Iterable, Sequence

import moderngl
import numpy as np
from colour import Color

from manim import config, logger
from manim.constants import *
from manim.utils.bezier import integer_interpolate, interpolate
from manim.utils.color import *
from manim.utils.color import Colors
from manim.utils.config_ops import _Data, _Uniforms

# from ..utils.iterables import batch_by_property
from manim.utils.iterables import (
    batch_by_property,
    list_update,
    listify,
    make_even,
    resize_array,
    resize_preserving_order,
    resize_with_interpolation,
    uniq_chain,
)
from manim.utils.paths import straight_path
from manim.utils.simple_functions import get_parameters
from manim.utils.space_ops import (
    angle_between_vectors,
    normalize,
    rotation_matrix_transpose,
)
)

def affects_shader_info_id(func):
    @wraps(func)
    def wrapper(self):
        for mob in self.get_family():
            func(mob)
            mob.refresh_shader_wrapper_id()
        return self

    return wrapper

class OpenGLObject:
    """Mathematical Object: base class for objects that can be displayed on screen.

    Attributes
    -----
    submobjects : List[:class:`OpenGLObject`]
        The contained objects.
    points : :class:`numpy.ndarray`
        The points of the objects.

    .. seealso::

        :class:`~OpenGLMobject`"""

    shader_dtype = [
        ("point", np.float32, (3,)),
    ]
    shader_folder = ""

    # _Data and _Uniforms are set as class variables to tell manim how to handle setting/getting these attributes later.
    points = _Data()
    bounding_box = _Data()
    rgbas = _Data()

    is_fixed_in_frame = _Uniforms()
    is_fixed_orientation = _Uniforms()
    fixed_orientation_center = _Uniforms() # for fixed orientation reference
    gloss = _Uniforms()
    shadow = _Uniforms()

    def __init__(
        self,
        color=WHITE,
```

```

opacity=1,
dim=3, # TODO, get rid of this
# Lighting parameters
# Positive gloss up to 1 makes it reflect the light.
gloss=0.0,
# Positive shadow up to 1 makes a side opposite the light darker
shadow=0.0,
# For shaders
render_primitive=moderngl.TRIANGLES,
texture_paths=None,
depth_test=False,
# If true, the mobject will not get rotated according to camera position
is_fixed_in_frame=False,
is_fixed_orientation=False,
# Must match in attributes of vert shader
# Event listener
listen_to_events=False,
model_matrix=None,
should_render=True,
name: str | None = None,
**kwargs,
):
    self.name = self.__class__.__name__ if name is None else name
    # setattr in case data/uniforms are already defined in parent classes.
    self.data = setattr(self, "data", {})
    self.uniforms = setattr(self, "uniforms", {})

    self.opacity = opacity
    self.dim = dim # TODO, get rid of this
    # Lighting parameters
    # Positive gloss up to 1 makes it reflect the light.
    self.gloss = gloss
    # Positive shadow up to 1 makes a side opposite the light darker
    self.shadow = shadow
    # For shaders
    self.render_primitive = render_primitive
    self.texture_paths = texture_paths
    self.depth_test = depth_test
    # If true, the mobject will not get rotated according to camera position
    self.is_fixed_in_frame = float(is_fixed_in_frame)
    self.is_fixed_orientation = float(is_fixed_orientation)
    self.fixed_orientation_center = (0, 0, 0)
    # Must match in attributes of vert shader
    # Event listener
    self.listen_to_events = listen_to_events

    self._subobjects = []
    self.parents = []
    self.parent = None
    self.family = [self]
    self.locked_data_keys = set()
    self.needs_new_bounding_box = True
    if model_matrix is None:
        self.model_matrix = np.eye(4)
    else:
        self.model_matrix = model_matrix

    self.init_data()
    self.init_updaters()
    # self.init_event_listners()
    self.init_points()
    self.color = Color(color) if color else None
    self.init_colors()

    self.shader_indices = None

    if self.depth_test:
        self.apply_depth_test()

    self.should_render = should_render

@classmethod
def __init_subclass__(cls, **kwargs):
    super().__init_subclass__(**kwargs)
    cls._original_init_ = cls.__init__

def __str__(self):
    return self.__class__.__name__

def __repr__(self):
    return str(self.name)

def __sub__(self, other):
    return NotImplemented

def __isub__(self, other):
    return NotImplemented

def __add__(self, mobject):
    return NotImplemented

def __iadd__(self, mobject):
    return NotImplemented

```

```

@classmethod
def set_default(cls, **kwargs):
    """Sets the default values of keyword arguments.

    If this method is called without any additional keyword
    arguments, the original default values of the initialization
    method of this class are restored.

    Parameters
    -----
    kwargs
        Passing any keyword argument will update the default
        values of the keyword arguments of the initialization
        function of this class.

    Examples
    -----
    ::

        >>> from manim import Square, GREEN
        >>> Square.set_default(color=GREEN, fill_opacity=0.25)
        >>> s = Square(); s.color, s.fill_opacity
        (<Color #83c167>, 0.25)
        >>> Square.set_default()
        >>> s = Square(); s.color, s.fill_opacity
        (<Color white>, 0.0)

    .. manim:: ChangedDefaultTextColor
        :save_last_frame:

        config.background_color = WHITE

        class ChangedDefaultTextColor(Scene):
            def construct(self):
                Text.set_default(color=BLACK)
                self.add(Text("Changing default values is easy!"))

                # we revert the colour back to the default to prevent a bug in the docs.
                Text.set_default(color=WHITE)

    """
    if kwargs:
        cls.__init__ = partialmethod(cls.__init__, **kwargs)
    else:
        cls.__init__ = cls._original_init__

    def init_data(self):
        """Initializes the ``points``, ``bounding_box`` and ``rgbas`` attributes and groups them into self.data.
        Subclasses can inherit and overwrite this method to extend `self.data`."""
        self.points = np.zeros((0, 3))
        self.bounding_box = np.zeros((3, 3))
        self.rgba = np.zeros((1, 4))

    def init_colors(self):
        """Initializes the colors.

        Gets called upon creation"""
        self.set_color(self.color, self.opacity)

    def init_points(self):
        """Initializes :attr:`points` and therefore the shape.

        Gets called upon creation. This is an empty method that can be implemented by
        subclasses."""
        # Typically implemented in subclass, unless purposefully left blank
        pass

    def set(self, **kwargs) -> OpenGLObject:
        """Sets attributes.

        Mainly to be used along with :attr:`animate` to
        animate setting attributes.

        Examples
        -----
        ::

            >>> mob = OpenGLObject()
            >>> mob.set(foo=0)
            OpenGLObject
            >>> mob.foo
            0

        Parameters
        -----
        **kwargs
            The attributes and corresponding values to set.

        Returns
        -----
    """

```

```

:class:`OpenGLMobject`  

    ``self``

"""

for attr, value in kwargs.items():
    setattr(self, attr, value)

return self

def set_data(self, data):
    for key in data:
        self.data[key] = data[key].copy()
    return self

def set_uniforms(self, uniforms):
    for key in uniforms:
        self.uniforms[key] = uniforms[key] # Copy?
    return self

@property
def animate(self):
    """Used to animate the application of a method.

.. warning::

    Passing multiple animations for the same :class:`OpenGLMobject` in one
    call to :meth:`~.Scene.play` is discouraged and will most likely
    not work properly. Instead of writing an animation like

    ::

        self.play(my_mobject.animate.shift(RIGHT), my_mobject.animate.rotate(PI))

    make use of method chaining for ``animate``, meaning::

        self.play(my_mobject.animate.shift(RIGHT).rotate(PI))

    Keyword arguments that can be passed to :meth:`.Scene.play` can be passed
    directly after accessing ``.animate``, like so::

        self.play(my_mobject.animate(rate_func=linear).shift(RIGHT))

    This is especially useful when animating simultaneous ``.animate`` calls that
    you want to behave differently::

        self.play(
            mobject1.animate(run_time=2).rotate(PI),
            mobject2.animate(rate_func=there_and_back).shift(RIGHT),
        )

.. seealso::

    :func:`override_animate`


Examples
-----
.. manim:: AnimateExample

    class AnimateExample(Scene):
        def construct(self):
            s = Square()
            self.play(Create(s))
            self.play(s.animate.shift(RIGHT))
            self.play(s.animate.scale(2))
            self.play(s.animate.rotate(PI / 2))
            self.play(Uncreate(s))

.. manim:: AnimateChainExample

    class AnimateChainExample(Scene):
        def construct(self):
            s = Square()
            self.play(Create(s))
            self.play(s.animate.shift(RIGHT).scale(2).rotate(PI / 2))
            self.play(Uncreate(s))

.. manim:: AnimateWithArgsExample

    class AnimateWithArgsExample(Scene):
        def construct(self):
            s = Square()
            c = Circle()

            VGroup(s, c).arrange(RIGHT, buff=2)
            self.add(s, c)

            self.play(
                s.animate(run_time=2).rotate(PI / 2),

```

```

        c.animate(rate_func=there_and_back).shift(RIGHT),
    )

.. warning::

    ```.animate```
 will interpolate the :class:`~OpenGLMobject` between its points prior to
    ```.animate``` and its points after applying ```.animate``` to it. This may
    result in unexpected behavior when attempting to interpolate along paths,
    or rotations.
    If you want animations to consider the points between, consider using
    :class:`~.ValueTracker` with updaters instead.

"""
return _AnimationBuilder(self)

@property
def width(self):
    """The width of the mobject.

    Returns
    ------
    :class:`float`

    Examples
    ------
    .. manim:: WidthExample

        class WidthExample(Scene):
            def construct(self):
                decimal = DecimalNumber().to_edge(UP)
                rect = Rectangle(color=BLUE)
                rect_copy = rect.copy().set_stroke(GRAY, opacity=0.5)

                decimal.add_updater(lambda d: d.set_value(rect.width))

                self.add(rect_copy, rect, decimal)
                self.play(rect.animate.set(width=7))
                self.wait()

    See also
    ------
    :meth:`length_over_dim`


    # Get the length across the X dimension
    return self.length_over_dim(0)

# Only these methods should directly affect points
@width.setter
def width(self, value):
    self.rescale_to_fit(value, 0, stretch=False)

@property
def height(self):
    """The height of the mobject.

    Returns
    ------
    :class:`float`

    Examples
    ------
    .. manim:: HeightExample

        class HeightExample(Scene):
            def construct(self):
                decimal = DecimalNumber().to_edge(UP)
                rect = Rectangle(color=BLUE)
                rect_copy = rect.copy().set_stroke(GRAY, opacity=0.5)

                decimal.add_updater(lambda d: d.set_value(rect.height))

                self.add(rect_copy, rect, decimal)
                self.play(rect.animate.set(height=5))
                self.wait()

    See also
    ------
    :meth:`length_over_dim`


    # Get the length across the Y dimension
    return self.length_over_dim(1)

@height.setter
def height(self, value):
    self.rescale_to_fit(value, 1, stretch=False)

@property
def depth(self):

```

```

    """The depth of the mobject.

Returns
-----
:class:`float`

See also
-----
:meth:`length_over_dim`


"""

# Get the length across the Z dimension
return self.length_over_dim(2)

@depth.setter
def depth(self, value):
    self.rescale_to_fit(value, 2, stretch=False)

def resize_points(self, new_length, resize_func=resize_array):
    if new_length != len(self.points):
        self.points = resize_func(self.points, new_length)
    self.refresh_bounding_box()
    return self

def set_points(self, points):
    if len(points) == len(self.points):
        self.points[:] = points
    elif isinstance(points, np.ndarray):
        self.points = points.copy()
    else:
        self.points = np.array(points)
    self.refresh_bounding_box()
    return self

def apply_over_attr_arrays(self, func):
    for attr in self.get_array_attrs():
        setattr(self, attr, func(getattr(self, attr)))
    return self

def append_points(self, new_points):
    self.points = np.vstack([self.points, new_points])
    self.refresh_bounding_box()
    return self

def reverse_points(self):
    for mob in self.get_family():
        for key in mob.data:
            mob.data[key] = mob.data[key][::-1]
    return self

def get_midpoint(self) -> np.ndarray:
    """Get coordinates of the middle of the path that forms the :class:`~.OpenGLMobject`."""

Examples
-----
.. manim:: AngleMidPoint
    :save_last_frame:

    class AngleMidPoint(Scene):
        def construct(self):
            line1 = Line(ORIGIN, 2*RIGHT)
            line2 = Line(ORIGIN, 2*RIGHT).rotate_about_origin(80*DEGREES)

            a = Angle(line1, line2, radius=1.5, other_angle=False)
            d = Dot(a.get_midpoint()).set_color(RED)

            self.add(line1, line2, a, d)
            self.wait()

    """
    return self.point_from_proportion(0.5)

def apply_points_function(
    self,
    func,
    about_point=None,
    about_edge=ORIGIN,
    works_on_bounding_box=False,
):
    if about_point is None and about_edge is not None:
        about_point = self.get_bounding_box_point(about_edge)

    for mob in self.get_family():
        arrs = []
        if mob.has_points():
            arrs.append(mob.points)
        if works_on_bounding_box:
            arrs.append(mob.get_bounding_box())

        for arr in arrs:
            if about_point is None:

```

```

        arr[:] = func(arr)
    else:
        arr[:] = func(arr - about_point) + about_point

    if not works_on_bounding_box:
        self.refresh_bounding_box(recurse_down=True)
    else:
        for parent in self.parents:
            parent.refresh_bounding_box()
    return self

# Others related to points

def match_points(self, mobject):
    """Edit points, positions, and submobjects to be identical
    to another :class:`~.OpenGMLObject`, while keeping the style unchanged.

Examples
-----
.. manim:: MatchPointsScene

    class MatchPointsScene(Scene):
        def construct(self):
            circ = Circle(fill_color=RED, fill_opacity=0.8)
            square = Square(fill_color=BLUE, fill_opacity=0.2)
            self.add(circ)
            self.wait(0.5)
            self.play(circ.animate.match_points(square))
            self.wait(0.5)

    """
    self.set_points(mobject.points)

def clear_points(self):
    self.points = np.empty((0, 3))

def get_num_points(self):
    return len(self.points)

def get_all_points(self):
    if self.submobjects:
        return np.vstack([sm.points for sm in self.get_family()])
    else:
        return self.points

def has_points(self):
    return self.get_num_points() > 0

def get_bounding_box(self):
    if self.needs_new_bounding_box:
        self.bounding_box = self.compute_bounding_box()
        self.needs_new_bounding_box = False
    return self.bounding_box

def compute_bounding_box(self):
    all_points = np.vstack(
        [
            self.points,
            *(
                mob.get_bounding_box()
                for mob in self.get_family()[1:]
                if mob.has_points()
            ),
        ],
    )
    if len(all_points) == 0:
        return np.zeros((3, self.dim))
    else:
        # Lower left and upper right corners
        mins = all_points.min(0)
        maxs = all_points.max(0)
        mids = (mins + maxs) / 2
        return np.array([mins, mids, maxs])

def refresh_bounding_box(self, recurse_down=False, recurse_up=True):
    for mob in self.get_family(recurse_down):
        mob.needs_new_bounding_box = True
    if recurse_up:
        for parent in self.parents:
            parent.refresh_bounding_box()
    return self

def is_point_touching(self, point, buff=MED_SMALL_BUFF):
    bb = self.get_bounding_box()
    mins = bb[0] - buff
    maxs = bb[2] + buff
    return (point >= mins).all() and (point <= maxs).all()

# Family matters

def __getitem__(self, value):
    if isinstance(value, slice):
        GroupClass = self.get_group_class()
        return GroupClass(*self.split().__getitem__(value))

```

```

    return self.split().__getitem__(value)

def __iter__(self):
    return iter(self.split())

def __len__(self):
    return len(self.split())

def split(self):
    return self.submobjects

def assemble_family(self):
    sub_families = (sm.get_family() for sm in self.submobjects)
    self.family = [self, *uniq_chain(*sub_families)]
    self.refresh_has_updater_status()
    self.refresh_bounding_box()
    for parent in self.parents:
        parent.assemble_family()
    return self

def get_family(self, recurse=True):
    if recurse and hasattr(self, "family"):
        return self.family
    else:
        return [self]

def family_members_with_points(self):
    return [m for m in self.get_family() if m.has_points()]

def add(
    self, *mobjects: OpenGLObject, update_parent: bool = False
) -> OpenGLObject:
    """Add mobjects as submobjects.

    The mobjects are added to :attr:`submobjects`.

    Subclasses of mobject may implement ``+`` and ``+=`` dunder methods.

    Parameters
    -----
    mobjects
        The mobjects to add.

    Returns
    -----
    :class:`OpenGLObject`
        ``self``

    Raises
    -----
    :class:`ValueError`
        When a mobject tries to add itself.
    :class:`TypeError`
        When trying to add an object that is not an instance of :class:`OpenGLObject`.

    Notes
    -----
    A mobject cannot contain itself, and it cannot contain a submobject more than once. If the parent mobject is displayed, the newly-added submobjects will also be displayed (i.e. they are automatically added to the parent Scene).

    See Also
    -----
    :meth:`remove`
    :meth:`add_to_back`

    Examples
    -----
    ::

        >>> outer = OpenGLObject()
        >>> inner = OpenGLObject()
        >>> outer = outer.add(inner)

   Duplicates are not added again::

        >>> outer = outer.add(inner)
        >>> len(outer.submobjects)
        1

    Adding an object to itself raises an error::

        >>> outer.add(outer)
        Traceback (most recent call last):
        ...
        ValueError: OpenGLObject cannot contain self

    """
    if update_parent:
        assert len(mobjects) == 1, "Can't set multiple parents."
        mobjects[0].parent = self

```

```

if self in mobjects:
    raise ValueError("OpenGLObject cannot contain self")
if any(mobjects.count(elem) > 1 for elem in mobjects):
    logger.warning(
        "Attempted adding some Mobject as a child more than once, "
        "this is not possible. Repetitions are ignored.",
    )
for mobject in mobjects:
    if not isinstance(mobject, OpenGLObject):
        raise TypeError("All submobjects must be of type OpenGLObject")
    if mobject not in self.submobjects:
        self.submobjects.append(mobject)
    if self not in mobject.parents:
        mobject.parents.append(self)
self.assemble_family()
return self

def insert(self, index: int, mobject: OpenGLObject, update_parent: bool = False):
    """Inserts a mobject at a specific position into self.submobjects

    Effectively just calls ``self.submobjects.insert(index, mobject)``,
    where ``self.submobjects`` is a list.

    Highly adapted from ``OpenGLObject.add``.

    Parameters
    -----
    index
        The index at which
    mobject
        The mobject to be inserted.
    update_parent
        Whether or not to set ``mobject.parent`` to ``self``.
    """

    if update_parent:
        mobject.parent = self

    if mobject is self:
        raise ValueError("OpenGLObject cannot contain self")

    if not isinstance(mobject, OpenGLObject):
        raise TypeError("All submobjects must be of type OpenGLObject")

    if mobject not in self.submobjects:
        self.submobjects.insert(index, mobject)

    if self not in mobject.parents:
        mobject.parents.append(self)

    self.assemble_family()
    return self

def remove(
    self, *mobjects: OpenGLObject, update_parent: bool = False
) -> OpenGLObject:
    """Remove :attr:`submobjects`.

    The mobjects are removed from :attr:`submobjects`, if they exist.

    Subclasses of mobject may implement ``-`` and ``-=`` dunder methods.

    Parameters
    -----
    mobjects
        The mobjects to remove.

    Returns
    -----
    :class:`OpenGLObject`
        ``self``

    See Also
    -----
    :meth:`add`

    """
    if update_parent:
        assert len(mobjects) == 1, "Can't remove multiple parents."
        mobjects[0].parent = None

    for mobject in mobjects:
        if mobject in self.submobjects:
            self.submobjects.remove(mobject)
        if self in mobject.parents:
            mobject.parents.remove(self)
    self.assemble_family()
    return self

def add_to_back(self, *mobjects: OpenGLObject) -> OpenGLObject:
    # NOTE: is the note true OpenGLObjects?
    """Add all passed mobjects to the back of the submobjects.

```

```

If :attr:`submobjects` already contains the given mobjects, they just get moved
to the back instead.

Parameters
-----
mobjects
    The mobjects to add.

Returns
-----
:class:`OpenGLMobject`
    ``self``

.. note::

    Technically, this is done by adding (or moving) the mobjects to
    the head of :attr:`submobjects`. The head of this list is rendered
    first, which places the corresponding mobjects behind the
    subsequent list members.

Raises
-----
:class:`ValueError`
    When a mobject tries to add itself.
:class:`TypeError`
    When trying to add an object that is not an instance of :class:`OpenGLMobject`.

Notes
-----
A mobject cannot contain itself, and it cannot contain a submobject
more than once. If the parent mobject is displayed, the newly-added
submobjects will also be displayed (i.e. they are automatically added
to the parent Scene).

See Also
-----
:meth:`remove`
:meth:`add`

"""
self.submobjects = list_update(mobjects, self.submobjects)
return self

def replace_submobject(self, index, new_submob):
    old_submob = self.submobjects[index]
    if self in old_submob.parents:
        old_submob.parents.remove(self)
    self.submobjects[index] = new_submob
    self.assemble_family()
    return self

def invert(self, recursive=False):
    """Inverts the list of :attr:`submobjects`.

Parameters
-----
recursive
    If ``True``, all submobject lists of this mobject's family are inverted.

Examples
-----
.. manim:: InvertSumobjectsExample

    class InvertSumobjectsExample(Scene):
        def construct(self):
            s = VGroup(*[Dot().shift(i*0.1*RIGHT) for i in range(-20,20)])
            s2 = s.copy()
            s2.invert()
            s2.shift(DOWN)
            self.play(Write(s), Write(s2))
    """
    if recursive:
        for submob in self.submobjects:
            submob.invert(recursive=True)
    list.reverse(self.submobjects)
    self.assemble_family()

# Submobject organization

def arrange(self, direction=RIGHT, center=True, **kwargs):
    """Sorts :class:`~OpenGLMobject` next to each other on screen.

Examples
-----
.. manim:: Example
    :save_last_frame:

    class Example(Scene):
        def construct(self):

```

```

        s1 = Square()
        s2 = Square()
        s3 = Square()
        s4 = Square()
        x = OpenGLGroup(s1, s2, s3, s4).set_x(0).arrange(buff=1.0)
        self.add(x)

"""
for m1, m2 in zip(self.submobjects, self.submobjects[1:]):
    m2.next_to(m1, direction, **kwargs)
if center:
    self.center()
return self

def arrange_in_grid(
    self,
    rows: int | None = None,
    cols: int | None = None,
    buff: float | tuple[float, float] = MED_SMALL_BUFF,
    cell_alignment: np.ndarray = ORIGIN,
    row_alignments: str | None = None, # "ucd"
    col_alignments: str | None = None, # "lcr"
    row_heights: Iterable[float | None] | None = None,
    col_widths: Iterable[float | None] | None = None,
    flow_order: str = "rd",
    **kwargs,
) -> OpenGLObject:
    """Arrange submobjects in a grid.

    Parameters
    -----
    rows
        The number of rows in the grid.
    cols
        The number of columns in the grid.
    buff
        The gap between grid cells. To specify a different buffer in the horizontal and
        vertical directions, a tuple of two values can be given - ````(row, col)````.
    cell_alignment
        The way each submobject is aligned in its grid cell.
    row_alignments
        The vertical alignment for each row (top to bottom). Accepts the following characters: ``"u"`` - up,
        ``"c"`` - center, ``"d"`` - down.
    col_alignments
        The horizontal alignment for each column (left to right). Accepts the following characters ``"l"`` - left,
        ``"c"`` - center, ``"r"`` - right.
    row_heights
        Defines a list of heights for certain rows (top to bottom). If the list contains
        ``None``, the corresponding row will fit its height automatically based
        on the highest element in that row.
    col_widths
        Defines a list of widths for certain columns (left to right). If the list contains ``None``, the
        corresponding column will fit its width automatically based on the widest element in that column.
    flow_order
        The order in which submobjects fill the grid. Can be one of the following values:
        "rd", "dr", "ld", "dl", "ru", "ur", "lu", "ul". ("rd" -> fill rightwards then downwards)

    Returns
    -----
    OpenGLObject
        The mobject.

    NOTES
    -----
    If only one of ``cols`` and ``rows`` is set implicitly, the other one will be chosen big
    enough to fit all submobjects. If neither is set, they will be chosen to be about the same,
    tending towards ``cols`` > ``rows`` (simply because videos are wider than they are high).

    If both ``cell_alignment`` and ``row_alignments`` / ``col_alignments`` are
    defined, the latter has higher priority.

    Raises
    -----
    ValueError
        If ``rows`` and ``cols`` are too small to fit all submobjects.
    ValueError
        If :code:`cols`, :code:`col_alignments` and :code:`col_widths` or :code:`rows`,
        :code:`row_alignments` and :code:`row_heights` have mismatching sizes.

    Examples
    -----
    .. manim:: ExampleBoxes
       :save_last_frame:

        class ExampleBoxes(Scene):
            def construct(self):
                boxes=VGroup(*[Square() for s in range(0,6)])
                boxes.arrange_in_grid(rows=2, buff=0.1)
                self.add(boxes)

    .. manim:: ArrangeInGrid

```

```

:save_last_frame:

class ArrangeInGrid(Scene):
    def construct(self):
        #Add some numbered boxes:
        np.random.seed(3)
        boxes = VGroup(*[
            Rectangle(WHITE, np.random.random() + .5, np.random.random() + .5).add(Text(str(i+1)).scale(0.5))
            for i in range(22)
        ])
        self.add(boxes)

        boxes.arrange_in_grid(
            buff=(0.25,0.5),
            col_alignments="lccccr",
            row_alignments="uccd",
            col_widths=[2, *[None]*4, 2],
            flow_order="dr"
        )

"""

from manim.mobject.geometry.line import Line

mobs = self.submobjects.copy()
start_pos = self.get_center()

# get cols / rows values if given (implicitly)
def init_size(num, alignments, sizes):
    if num is not None:
        return num
    if alignments is not None:
        return len(alignments)
    if sizes is not None:
        return len(sizes)

cols = init_size(cols, col_alignments, col_widths)
rows = init_size(rows, row_alignments, row_heights)

# calculate rows cols
if rows is None and cols is None:
    cols = ceil(np.sqrt(len(mobs)))
    # make the grid as close to quadratic as possible.
    # choosing cols first can results in cols>rows.
    # This is favored over rows>cols since in general
    # the scene is wider than high.
if rows is None:
    rows = ceil(len(mobs) / cols)
if cols is None:
    cols = ceil(len(mobs) / rows)
if rows * cols < len(mobs):
    raise ValueError("Too few rows and columns to fit all submobjetc")
# rows and cols are now finally valid.

if isinstance(buff, tuple):
    buff_x = buff[0]
    buff_y = buff[1]
else:
    buff_x = buff_y = buff

# Initialize alignments correctly
def init_alignments(alignments, num, mapping, name, dir):
    if alignments is None:
        # Use cell_alignment as fallback
        return [cell_alignment * dir] * num
    if len(alignments) != num:
        raise ValueError(f"{name}_alignments has a mismatching size.")
    alignments = list(alignments)
    for i in range(num):
        alignments[i] = mapping[alignments[i]]
    return alignments

row_alignments = init_alignments(
    col_alignments,
    rows,
    {"u": UP, "c": ORIGIN, "d": DOWN},
    "row",
    RIGHT,
)
col_alignments = init_alignments(
    col_alignments,
    cols,
    {"l": LEFT, "c": ORIGIN, "r": RIGHT},
    "col",
    UP,
)
# Now row_alignment[r] + col_alignment[c] is the alignment in cell [r][c]

mapper = {
    "dr": lambda r, c: (rows - r - 1) + c * rows,
    "dl": lambda r, c: (rows - r - 1) + (cols - c - 1) * rows,
    "ur": lambda r, c: r + c * rows,
    "ul": lambda r, c: r + (cols - c - 1) * rows,
}

```

```

    "rd": lambda r, c: (rows - r - 1) * cols + c,
    "ld": lambda r, c: (rows - r - 1) * cols + (cols - c - 1),
    "ru": lambda r, c: r * cols + c,
    "lu": lambda r, c: r * cols + (cols - c - 1),
}
if flow_order not in mapper:
    raise ValueError(
        'flow_order must be one of the following values: "dr", "rd", "ld" "dl", "ru", "ur", "lu", "ul".',
    )
flow_order = mapper[flow_order]

# Reverse row_alignments and row_heights. Necessary since the
# grid filling is handled bottom up for simplicity reasons.
def reverse(maybe_list):
    if maybe_list is not None:
        maybe_list = list(maybe_list)
        maybe_list.reverse()
    return maybe_list

row_alignments = reverse(row_alignments)
row_heights = reverse(row_heights)

placeholder = OpenGLMObject()
# Used to fill up the grid temporarily, doesn't get added to the scene.
# In this case a Mobject is better than None since it has width and height
# properties of 0.

mobs.extend([placeholder] * (rows * cols - len(mobs)))
grid = [[mobs[flow_order(r, c)] for c in range(cols)] for r in range(rows)]

measured_heights = [
    max(grid[r][c].height for c in range(cols)) for r in range(rows)
]
measured_widths = [
    max(grid[r][c].width for r in range(rows)) for c in range(cols)
]

# Initialize row_heights / col_widths correctly using measurements as fallback
def init_sizes(sizes, num, measures, name):
    if sizes is None:
        sizes = [None] * num
    if len(sizes) != num:
        raise ValueError(f"{name} has a mismatching size.")
    return [
        sizes[i] if sizes[i] is not None else measures[i] for i in range(num)
    ]

heights = init_sizes(row_heights, rows, measured_heights, "row_heights")
widths = init_sizes(col_widths, cols, measured_widths, "col_widths")

x, y = 0, 0
for r in range(rows):
    x = 0
    for c in range(cols):
        if grid[r][c] is not placeholder:
            alignment = row_alignments[r] + col_alignments[c]
            line = Line(
                x * RIGHT + y * UP,
                (x + widths[c]) * RIGHT + (y + heights[r]) * UP,
            )
            # Use a mobject to avoid rewriting align inside
            # box code that Mobject.move_to(Mobject) already
            # includes.

            grid[r][c].move_to(line, alignment)
            x += widths[c] + buff_x
            y += heights[r] + buff_y

    self.move_to(start_pos)
    return self

def get_grid(self, n_rows, n_cols, height=None, **kwargs):
    """
    Returns a new mobject containing multiple copies of this one
    arranged in a grid
    """
    grid = self.duplicate(n_rows * n_cols)
    grid.arrange_in_grid(n_rows, n_cols, **kwargs)
    if height is not None:
        grid.set_height(height)
    return grid

def duplicate(self, n: int):
    """Returns an :class:`~.OpenGLVGGroup` containing ``n`` copies of the mobject."""
    return self.get_group_class()(*[self.copy() for _ in range(n)])

def sort(self, point_to_num_func=lambda p: p[0], submob_func=None):
    """Sorts the list of :attr:`submobjects` by a function defined by ``submob_func``."""
    if submob_func is not None:
        self.submobjects.sort(key=submob_func)
    else:
        self.submobjects.sort(key=lambda m: point_to_num_func(m.get_center()))
    return self

```

```

def shuffle(self, recurse=False):
    """Shuffles the order of :attr:`submobjects`"""

Examples
-----
.. manim:: ShuffleSubmobjectsExample

    class ShuffleSubmobjectsExample(Scene):
        def construct(self):
            s= OpenGLGroup(*[Dot().shift(i*0.1*RIGHT) for i in range(-20,20)])
            s2= s.copy()
            s2.shuffle()
            s2.shift(DOWN)
            self.play(Write(s), Write(s2))

    """
    if recurse:
        for submob in self.submobjects:
            submob.shuffle(recurse=True)
    random.shuffle(self.submobjects)
    self.assemble_family()
    return self

def invert(self, recursive=False):
    """Inverts the list of :attr:`submobjects`."""

Parameters
-----
recursive
    If ``True``, all submobject lists of this mobject's family are inverted.

Examples
-----
.. manim:: InvertSumobjectsExample

    class InvertSumobjectsExample(Scene):
        def construct(self):
            s = VGroup(*[Dot().shift(i*0.1*RIGHT) for i in range(-20,20)])
            s2 = s.copy()
            s2.invert()
            s2.shift(DOWN)
            self.play(Write(s), Write(s2))

    """
    if recursive:
        for submob in self.submobjects:
            submob.invert(recursive=True)
        list.reverse(self.submobjects)

# Copying

def copy(self, shallow: bool = False):
    """Create and return an identical copy of the :class:`OpenGLMobject` including all :attr:`submobjects`."""

Returns
-----
:class:`OpenGLMobject`
    The copy.

Parameters
-----
shallow
    Controls whether a shallow copy is returned.

Note
----
The clone is initially not visible in the Scene, even if the original was.

    """
if not shallow:
    return self.deepcopy()

# TODO, either justify reason for shallow copy, or
# remove this redundancy everywhere
# return self.deepcopy()

parents = self.parents
self.parents = []
copy_mobject = copy.deepcopy(self)
self.parents = parents

copy_mobject.data = dict(self.data)
for key in self.data:
    copy_mobject.data[key] = self.data[key].copy()

# TODO, are uniforms ever numpy arrays?
copy_mobject.uniforms = dict(self.uniforms)

copy_mobject.submobjects = []
copy_mobject.add(*([sm.copy() for sm in self.submobjects]))
copy_mobject.match_updaters(self)

```

```

copy_mobject.needs_new_bounding_box = self.needs_new_bounding_box

# Make sure any mobobject or numpy array attributes are copied
family = self.get_family()
for attr, value in list(self.__dict__.items()):
    if (
        isinstance(value, OpenGLMobject)
        and value in family
        and value is not self
    ):
        setattr(copy_mobject, attr, value.copy())
    if isinstance(value, np.ndarray):
        setattr(copy_mobject, attr, value.copy())
    # if isinstance(value, ShaderWrapper):
    #     setattr(copy_mobject, attr, value.copy())
return copy_mobject

def deepcopy(self):
    parents = self.parents
    self.parents = []
    result = copy.deepcopy(self)
    self.parents = parents
    return result

def generate_target(self, use_deepcopy: bool = False):
    self.target = None # Prevent exponential explosion
    if use_deepcopy:
        self.target = self.deepcopy()
    else:
        self.target = self.copy()
    return self.target

def save_state(self, use_deepcopy: bool = False):
    """Save the current state (position, color & size). Can be restored with :meth:`~OpenGLMobject.restore`."""
    if hasattr(self, "saved_state"):
        # Prevent exponential growth of data
        self.saved_state = None
    if use_deepcopy:
        self.saved_state = self.deepcopy()
    else:
        self.saved_state = self.copy()
    return self

def restore(self):
    """Restores the state that was previously saved with :meth:`~OpenGLMobject.save_state`."""
    if not hasattr(self, "saved_state") or self.saved_state is None:
        raise Exception("Trying to restore without having saved")
    self.become(self.saved_state)
    return self

# Updating

def init_updaters(self):
    self.time_based_updaters = []
    self.non_time_updaters = []
    self.has_updaters = False
    self.updating_suspended = False

def update(self, dt=0, recurse=True):
    if not self.has_updaters or self.updating_suspended:
        return self
    for updater in self.time_based_updaters:
        updater(self, dt)
    for updater in self.non_time_updaters:
        updater(self)
    if recurse:
        for submob in self.submobjects:
            submob.update(dt, recurse)
    return self

def get_time_based_updaters(self):
    return self.time_based_updaters

def has_time_based_updater(self):
    return len(self.time_based_updaters) > 0

def get_updaters(self):
    return self.time_based_updaters + self.non_time_updaters

def get_family_updaters(self):
    return list(it.chain(*(sm.get_updaters() for sm in self.get_family())))

def add_updater(self, update_function, index=None, call_updater=False):
    if "dt" in get_parameters(update_function):
        updater_list = self.time_based_updaters
    else:
        updater_list = self.non_time_updaters

    if index is None:
        updater_list.append(update_function)
    else:
        updater_list.insert(index, update_function)

```

```

self.refresh_has_updater_status()
if call_updater:
    self.update()
return self

def remove_updater(self, update_function):
    for updater_list in [self.time_based_updaters, self.non_time_updaters]:
        while update_function in updater_list:
            updater_list.remove(update_function)
    self.refresh_has_updater_status()
    return self

def clear_updaters(self, recurse=True):
    self.time_based_updaters = []
    self.non_time_updaters = []
    self.refresh_has_updater_status()
    if recurse:
        for submob in self.submobjects:
            submob.clear_updaters()
    return self

def match_updaters(self, mobject):
    self.clear_updaters()
    for updater in mobject.get_updaters():
        self.add_updater(updater)
    return self

def suspend_updating(self, recurse=True):
    self.updating_suspended = True
    if recurse:
        for submob in self.submobjects:
            submob.suspend_updating(recurse)
    return self

def resume_updating(self, recurse=True, call_updater=True):
    self.updating_suspended = False
    if recurse:
        for submob in self.submobjects:
            submob.resume_updating(recurse)
    for parent in self.parents:
        parent.resume_updating(recurse=False, call_updater=False)
    if call_updater:
        self.update(dt=0, recurse=recurse)
    return self

def refresh_has_updater_status(self):
    self.has_updaters = any(mob.get_updaters() for mob in self.get_family())
    return self

# Transforming operations

def shift(self, vector):
    self.apply_points_function(
        lambda points: points + vector,
        about_edge=None,
        works_on_bounding_box=True,
    )
    return self

def scale(
    self,
    scale_factor: float,
    about_point: Sequence[float] | None = None,
    about_edge: Sequence[float] = ORIGIN,
    **kwargs,
) -> OpenGLMobject:
    r"""Scale the size by a factor.

    Default behavior is to scale about the center of the mobject.
    The argument about_edge can be a vector, indicating which side of
    the mobject to scale about, e.g., mob.scale(about_edge = RIGHT)
    scales about mob.get_right().

    Otherwise, if about_point is given a value, scaling is done with
    respect to that point.

    Parameters
    -----
    scale_factor
        The scaling factor :math:`\alpha`. If :math:`0 < |\alpha| < 1`, the mobject
        will shrink, and for :math:`|\alpha| > 1` it will grow. Furthermore,
        if :math:`\alpha < 0`, the mobject is also flipped.
    kwargs
        Additional keyword arguments passed to
        :meth:`apply_points_function_about_point`.

    Returns
    -----
    OpenGLMobject
        The scaled mobject.

    Examples
    -----
    """

```

```

.. manim:: MobjectScaleExample
    :save_last_frame:

    class MobjectScaleExample(Scene):
        def construct(self):
            f1 = Text("F")
            f2 = Text("F").scale(2)
            f3 = Text("F").scale(0.5)
            f4 = Text("F").scale(-1)

            vgroup = VGroup(f1, f2, f3, f4).arrange(6 * RIGHT)
            self.add(vgroup)

    See also
    ======
    :meth:`move_to`


    """
    self.apply_points_function(
        lambda points: scale_factor * points,
        about_point=about_point,
        about_edge=about_edge,
        works_on_bounding_box=True,
        **kwargs,
    )
    return self

def stretch(self, factor, dim, **kwargs):
    def func(points):
        points[:, dim] *= factor
        return points

    self.apply_points_function(func, works_on_bounding_box=True, **kwargs)
    return self

def rotate_about_origin(self, angle, axis=OUT):
    return self.rotate(angle, axis, about_point=ORIGIN)

def rotate(
    self,
    angle,
    axis=OUT,
    about_point: Sequence[float] | None = None,
    **kwargs,
) :
    """Rotates the :class:`~.OpenGLMobject` about a certain point."""
    rot_matrix_T = rotation_matrix_transpose(angle, axis)
    self.apply_points_function(
        lambda points: np.dot(points, rot_matrix_T),
        about_point=about_point,
        **kwargs,
    )
    return self

def flip(self, axis=UP, **kwargs):
    """Flips/Mirrors an mobject about its center.

    Examples
    ======

    .. manim:: FlipExample
        :save_last_frame:

        class FlipExample(Scene):
            def construct(self):
                s= Line(LEFT, RIGHT+UP).shift(4*LEFT)
                self.add(s)
                s2= s.copy().flip()
                self.add(s2)

    """
    return self.rotate(TAU / 2, axis, **kwargs)

def apply_function(self, function, **kwargs):
    # Default to applying matrix about the origin, not mobjects center
    if len(kwargs) == 0:
        kwargs["about_point"] = ORIGIN
    self.apply_points_function(
        lambda points: np.array([function(p) for p in points]), **kwargs
    )
    return self

def apply_function_to_position(self, function):
    self.move_to(function(self.get_center()))
    return self

def apply_function_to_submobject_positions(self, function):
    for submob in self.submobjects:
        submob.apply_function_to_position(function)
    return self

def apply_matrix(self, matrix, **kwargs):

```

```

# Default to applying matrix about the origin, not mobjects center
if ("about_point" not in kwargs) and ("about_edge" not in kwargs):
    kwargs["about_point"] = ORIGIN
full_matrix = np.identity(self.dim)
matrix = np.array(matrix)
full_matrix[: matrix.shape[0], : matrix.shape[1]] = matrix
self.apply_points_function(
    lambda points: np.dot(points, full_matrix.T), **kwargs
)
return self

def apply_complex_function(self, function, **kwargs):
    """Applies a complex function to a :class:`OpenGLMobject`.  

    The x and y coordinates correspond to the real and imaginary parts respectively.

Example
-----
.. manim:: ApplyFuncExample

class ApplyFuncExample(Scene):
    def construct(self):
        circ = Circle().scale(1.5)
        circ_ref = circ.copy()
        circ.apply_complex_function(
            lambda x: np.exp(x*1j)
        )
        t = ValueTracker(0)
        circ.add_updater(
            lambda x: x.become(circ_ref.copy().apply_complex_function(
                lambda x: np.exp(x+t.get_value()*1j)
            )).set_color(BLUE)
        )
        self.add(circ_ref)
        self.play(TransformFromCopy(circ_ref, circ))
        self.play(t.animate.set_value(TAU), run_time=3)
"""

def R3_func(point):
    x, y, z = point
    xy_complex = function(complex(x, y))
    return [xy_complex.real, xy_complex.imag, z]

return self.apply_function(R3_func)

def hierarchical_model_matrix(self):
    if self.parent is None:
        return self.model_matrix

    model_matrices = [self.model_matrix]
    current_object = self
    while current_object.parent is not None:
        model_matrices.append(current_object.parent.model_matrix)
        current_object = current_object.parent
    return np.linalg.multi_dot(list(reversed(model_matrices)))

def wag(self, direction=RIGHT, axis=DOWN, wag_factor=1.0):
    for mob in self.family_members_with_points():
        alphas = np.dot(mob.points, np.transpose(axis))
        alphas -= min(alphas)
        alphas /= max(alphas)
        alphas = alphas**wag_factor
        mob.set_points(
            mob.points
            + np.dot(
                alphas.reshape((len(alphas), 1)),
                np.array(direction).reshape((1, mob.dim)),
            ),
        )
    return self

# Positioning methods

def center(self):
    """Moves the mobject to the center of the Scene."""
    self.shift(-self.get_center())
    return self

def align_on_border(self, direction, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
    """
    Direction just needs to be a vector pointing towards side or
    corner in the 2d plane.
    """
    target_point = np.sign(direction) * (
        config["frame_x_radius"],
        config["frame_y_radius"],
        0,
    )
    point_to_align = self.get_bounding_box_point(direction)
    shift_val = target_point - point_to_align - buff * np.array(direction)
    shift_val = shift_val * abs(np.sign(direction))
    self.shift(shift_val)
    return self

```

```

def to_corner(self, corner=LEFT + DOWN, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
    return self.align_on_border(corner, buff)

def to_edge(self, edge=LEFT, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
    return self.align_on_border(edge, buff)

def next_to(
    self,
    mobobject_or_point,
    direction=RIGHT,
    buff=DEFAULT_MOBJECT_TO_MOBJECT_BUFFER,
    aligned_edge=ORIGIN,
    submobobject_to_align=None,
    index_of_submobobject_to_align=None,
    coor_mask=np.array([1, 1, 1]),
):
    """Move this :class:`~.OpenGLMobject` next to another's :class:`~.OpenGLMobject` or coordinate.

Examples
-----
.. manim:: GeometricShapes
:save_last_frame:

class GeometricShapes(Scene):
    def construct(self):
        d = Dot()
        c = Circle()
        s = Square()
        t = Triangle()
        d.next_to(c, RIGHT)
        s.next_to(c, LEFT)
        t.next_to(c, DOWN)
        self.add(d, c, s, t)

"""
    if isinstance(mobobject_or_point, OpenGLMobject):
        mob = mobobject_or_point
        if index_of_submobobject_to_align is not None:
            target_aligner = mob[index_of_submobobject_to_align]
        else:
            target_aligner = mob
        target_point = target_aligner.get_bounding_box_point(
            aligned_edge + direction,
        )
    else:
        target_point = mobobject_or_point
    if submobobject_to_align is not None:
        aligner = submobobject_to_align
    elif index_of_submobobject_to_align is not None:
        aligner = self[index_of_submobobject_to_align]
    else:
        aligner = self
    point_to_align = aligner.get_bounding_box_point(aligned_edge - direction)
    self.shift((target_point - point_to_align + buff * direction) * coor_mask)
    return self

def shift_onto_screen(self, **kwargs):
    space_lengths = [config["frame_x_radius"], config["frame_y_radius"]]
    for vect in UP, DOWN, LEFT, RIGHT:
        dim = np.argmax(np.abs(vect))
        buff = kwargs.get("buff", DEFAULT_MOBJECT_TO_EDGE_BUFFER)
        max_val = space_lengths[dim] - buff
        edge_center = self.get_edge_center(vect)
        if np.dot(edge_center, vect) > max_val:
            self.to_edge(vect, **kwargs)
    return self

def is_off_screen(self):
    if self.get_left()[0] > config.frame_x_radius:
        return True
    if self.get_right()[0] < config.frame_x_radius:
        return True
    if self.get_bottom()[1] > config.frame_y_radius:
        return True
    if self.get_top()[1] < -config.frame_y_radius:
        return True
    return False

def stretch_about_point(self, factor, dim, point):
    return self.stretch(factor, dim, about_point=point)

def rescale_to_fit(self, length, dim, stretch=False, **kwargs):
    old_length = self.length_over_dim(dim)
    if old_length == 0:
        return self
    if stretch:
        self.stretch(length / old_length, dim, **kwargs)
    else:
        self.scale(length / old_length, **kwargs)
    return self

```

```

def stretch_to_fit_width(self, width, **kwargs):
    """Stretches the :class:`~.OpenGLMobject` to fit a width, not keeping height/depth proportional.

    Returns
    ------
    :class:`OpenGLMobject`
        ``self``

    Examples
    ------
    ::

        >>> from manim import *
        >>> sq = Square()
        >>> sq.height
        2.0
        >>> sq.stretch_to_fit_width(5)
        Square
        >>> sq.width
        5.0
        >>> sq.height
        2.0
    """
    return self.rescale_to_fit(width, 0, stretch=True, **kwargs)

def stretch_to_fit_height(self, height, **kwargs):
    """Stretches the :class:`~.OpenGLMobject` to fit a height, not keeping width/height proportional."""
    return self.rescale_to_fit(height, 1, stretch=True, **kwargs)

def stretch_to_fit_depth(self, depth, **kwargs):
    """Stretches the :class:`~.OpenGLMobject` to fit a depth, not keeping width/height proportional."""
    return self.rescale_to_fit(depth, 1, stretch=True, **kwargs)

def set_width(self, width, stretch=False, **kwargs):
    """Scales the :class:`~.OpenGLMobject` to fit a width while keeping height/depth proportional.

    Returns
    ------
    :class:`OpenGLMobject`
        ``self``

    Examples
    ------
    ::

        >>> from manim import *
        >>> sq = Square()
        >>> sq.height
        2.0
        >>> sq.scale_to_fit_width(5)
        Square
        >>> sq.width
        5.0
        >>> sq.height
        5.0
    """
    return self.rescale_to_fit(width, 0, stretch=stretch, **kwargs)

scale_to_fit_width = set_width

def set_height(self, height, stretch=False, **kwargs):
    """Scales the :class:`~.OpenGLMobject` to fit a height while keeping width/depth proportional."""
    return self.rescale_to_fit(height, 1, stretch=stretch, **kwargs)

scale_to_fit_height = set_height

def set_depth(self, depth, stretch=False, **kwargs):
    """Scales the :class:`~.OpenGLMobject` to fit a depth while keeping width/height proportional."""
    return self.rescale_to_fit(depth, 2, stretch=stretch, **kwargs)

scale_to_fit_depth = set_depth

def set_coord(self, value, dim, direction=ORIGIN):
    curr = self.get_coord(dim, direction)
    shift_vect = np.zeros(self.dim)
    shift_vect[dim] = value - curr
    self.shift(shift_vect)
    return self

def set_x(self, x, direction=ORIGIN):
    """Set x value of the center of the :class:`~.OpenGLMobject` ('`int`' or '`float`')"""
    return self.set_coord(x, 0, direction)

def set_y(self, y, direction=ORIGIN):
    """Set y value of the center of the :class:`~.OpenGLMobject` ('`int`' or '`float`')"""
    return self.set_coord(y, 1, direction)

def set_z(self, z, direction=ORIGIN):
    """Set z value of the center of the :class:`~.OpenGLMobject` ('`int`' or '`float`')"""
    return self.set_coord(z, 2, direction)

def space_out_submobjects(self, factor=1.5, **kwargs):
    self.scale(factor, **kwargs)

```

```

for submob in self.submobjects:
    submob.scale(1.0 / factor)
return self

def move_to(
    self,
    point_or_mobject,
    aligned_edge=ORIGIN,
    coor_mask=np.array([1, 1, 1]),
):
    """Move center of the :class:`~.OpenGLMobject` to certain coordinate."""
    if isinstance(point_or_mobject, OpenGLMobject):
        target = point_or_mobject.get_bounding_box_point(aligned_edge)
    else:
        target = point_or_mobject
    point_to_align = self.get_bounding_box_point(aligned_edge)
    self.shift((target - point_to_align) * coor_mask)
    return self

def replace(self, mobject, dim_to_match=0, stretch=False):
    if not mobject.get_num_points() and not mobject.submobjects:
        self.scale(0)
        return self
    if stretch:
        for i in range(self.dim):
            self.rescale_to_fit(mobject.length_over_dim(i), i, stretch=True)
    else:
        self.rescale_to_fit(
            mobject.length_over_dim(dim_to_match),
            dim_to_match,
            stretch=False,
        )
    self.shift(mobject.get_center() - self.get_center())
    return self

def surround(
    self,
    mobject: OpenGLMobject,
    dim_to_match: int = 0,
    stretch: bool = False,
    buff: float = MED_SMALL_BUFF,
):
    self.replace(mobject, dim_to_match, stretch)
    length = mobject.length_over_dim(dim_to_match)
    self.scale((length + buff) / length)
    return self

def put_start_and_end_on(self, start, end):
    curr_start, curr_end = self.get_start_and_end()
    curr_vect = curr_end - curr_start
    if np.all(curr_vect == 0):
        raise Exception("Cannot position endpoints of closed loop")
    target_vect = np.array(end) - np.array(start)
    axis = (
        normalize(np.cross(curr_vect, target_vect))
        if np.linalg.norm(np.cross(curr_vect, target_vect)) != 0
        else OUT
    )
    self.scale(
        np.linalg.norm(target_vect) / np.linalg.norm(curr_vect),
        about_point=curr_start,
    )
    self.rotate(
        angle_between_vectors(curr_vect, target_vect),
        about_point=curr_start,
        axis=axis,
    )
    self.shift(start - curr_start)
    return self

# Color functions

def set_rgba_array(self, color=None, opacity=None, name="rgbas", recurse=True):
    if color is not None:
        rgbs = np.array([color_to_rgb(c) for c in listify(color)])
    if opacity is not None:
        opacities = listify(opacity)

    # Color only
    if color is not None and opacity is None:
        for mob in self.get_family(recurse):
            mob.data[name] = resize_array(
                mob.data[name] if name in mob.data else np.empty((1, 3)),
                len(rgbs)
            )
            mob.data[name][:, :3] = rgbs

    # Opacity only
    if color is None and opacity is not None:
        for mob in self.get_family(recurse):
            mob.data[name] = resize_array(
                mob.data[name] if name in mob.data else np.empty((1, 3)),
                len(opacities),
            )

```

```

        mob.data[name][:, 3] = opacities

    # Color and opacity
    if color is not None and opacity is not None:
        rgbas = np.array([*[rgb, o] for rgb, o in zip(*make_even(rgbs, opacities))])
        for mob in self.get_family(recurse):
            mob.data[name] = rgbas.copy()
    return self

def set_rgba_array_direct(self, rgbas: np.ndarray, name="rgbas", recurse=True):
    """Directly set rgba data from `rgbas` and optionally do the same recursively
    with subobjects. This can be used if the `rgbas` have already been generated
    with the correct shape and simply need to be set.

Parameters
-----
rgbas
    the rgba to be set as data
name
    the name of the data attribute to be set
recurse
    set to true to recursively apply this method to subobjects
"""
    for mob in self.get_family(recurse):
        mob.data[name] = rgbas.copy()

def set_color(self, color, opacity=None, recurse=True):
    self.set_rgba_array(color, opacity, recurse=False)
    # Recurse to subobjects differently from how set_rgba_array
    # in case they implement set_color differently
    if color is not None:
        self.color = Color(color)
    if opacity is not None:
        self.opacity = opacity
    if recurse:
        for submob in self.subobjects:
            submob.set_color(color, recurse=True)
    return self

def set_opacity(self, opacity, recurse=True):
    self.set_rgba_array(color=None, opacity=opacity, recurse=False)
    if recurse:
        for submob in self.subobjects:
            submob.set_opacity(opacity, recurse=True)
    return self

def get_color(self):
    return rgb_to_hex(self.rgba[0, :3])

def get_opacity(self):
    return self.opacity

def set_color_by_gradient(self, *colors):
    self.set_subobject_colors_by_gradient(*colors)
    return self

def set_subobject_colors_by_gradient(self, *colors):
    if len(colors) == 0:
        raise Exception("Need at least one color")
    elif len(colors) == 1:
        return self.set_color(*colors)

    # mobs = self.family_members_with_points()
    mobs = self.subobjects
    new_colors = color_gradient(colors, len(mobs))

    for mob, color in zip(mobs, new_colors):
        mob.set_color(color)
    return self

def fade(self, darkness=0.5, recurse=True):
    self.set_opacity(1.0 - darkness, recurse=recurse)

def get_gloss(self):
    return self.gloss

def set_gloss(self, gloss, recurse=True):
    for mob in self.get_family(recurse):
        mob.gloss = gloss
    return self

def get_shadow(self):
    return self.shadow

def set_shadow(self, shadow, recurse=True):
    for mob in self.get_family(recurse):
        mob.shadow = shadow
    return self

# Background rectangle

def add_background_rectangle(
    self, color: Colors | None = None, opacity: float = 0.75, **kwargs

```

```

):
    # TODO, this does not behave well when the mobject has points,
    # since it gets displayed on top
    """Add a BackgroundRectangle as submobject.

The BackgroundRectangle is added behind other submobjects.

This can be used to increase the mobjects visibility in front of a noisy background.

Parameters
-----
color
    The color of the BackgroundRectangle
opacity
    The opacity of the BackgroundRectangle
kwargs
    Additional keyword arguments passed to the BackgroundRectangle constructor

Returns
-----
:class:`~OpenGLMobject`
``self``

See Also
-----
:meth:`add_to_back`
:class:`~.BackgroundRectangle`


"""
from manim.mobject.geometry.shape_matchers import BackgroundRectangle

self.background_rectangle = BackgroundRectangle(
    self, color=color, fill_opacity=opacity, **kwargs
)
self.add_to_back(self.background_rectangle)
return self

def add_background_rectangle_to_submobjects(self, **kwargs):
    for submobject in self.submobjects:
        submobject.add_background_rectangle(**kwargs)
    return self

def add_background_rectangle_to_family_members_with_points(self, **kwargs):
    for mob in self.family_members_with_points():
        mob.add_background_rectangle(**kwargs)
    return self

# Getters

def get_bounding_box_point(self, direction):
    bb = self.get_bounding_box()
    indices = (np.sign(direction) + 1).astype(int)
    return np.array([bb[indices[i]][i] for i in range(3)])

def get_edge_center(self, direction) -> np.ndarray:
    """Get edge coordinates for certain direction."""
    return self.get_bounding_box_point(direction)

def get_corner(self, direction) -> np.ndarray:
    """Get corner coordinates for certain direction."""
    return self.get_bounding_box_point(direction)

def get_center(self) -> np.ndarray:
    """Get center coordinates."""
    return self.get_bounding_box()[1]

def get_center_of_mass(self):
    return self.get_all_points().mean(0)

def get_boundary_point(self, direction):
    all_points = self.get_all_points()
    boundary_directions = all_points - self.get_center()
    norms = np.linalg.norm(boundary_directions, axis=1)
    boundary_directions /= np.repeat(norms, 3).reshape((len(norms), 3))
    index = np.argmax(np.dot(boundary_directions, np.array(direction).T))
    return all_points[index]

def get_continuous_bounding_box_point(self, direction):
    d1, center, ur = self.get_bounding_box()
    corner_vect = ur - center
    return center + direction / np.max(
        np.abs(
            np.true_divide(
                direction,
                corner_vect,
                out=np.zeros(len(direction)),
                where=((corner_vect) != 0),
            ),
            ),
    )
    )

def get_top(self) -> np.ndarray:

```

```

"""Get top coordinates of a box bounding the :class:`~.OpenGLMobject`"""
return self.get_edge_center(UP)

def get_bottom(self) -> np.ndarray:
    """Get bottom coordinates of a box bounding the :class:`~.OpenGLMobject`"""
    return self.get_edge_center(DOWN)

def get_right(self) -> np.ndarray:
    """Get right coordinates of a box bounding the :class:`~.OpenGLMobject`"""
    return self.get_edge_center(RIGHT)

def get_left(self) -> np.ndarray:
    """Get left coordinates of a box bounding the :class:`~.OpenGLMobject`"""
    return self.get_edge_center(LEFT)

def get_zenith(self) -> np.ndarray:
    """Get zenith coordinates of a box bounding a 3D :class:`~.OpenGLMobject`."""
    return self.get_edge_center(OUT)

def get_nadir(self) -> np.ndarray:
    """Get nadir (opposite the zenith) coordinates of a box bounding a 3D :class:`~.OpenGLMobject`."""
    return self.get_edge_center(IN)

def length_over_dim(self, dim):
    bb = self.get_bounding_box()
    return abs((bb[2] - bb[0])[dim])

def get_width(self):
    """Returns the width of the mobject."""
    return self.length_over_dim(0)

def get_height(self):
    """Returns the height of the mobject."""
    return self.length_over_dim(1)

def get_depth(self):
    """Returns the depth of the mobject."""
    return self.length_over_dim(2)

def get_coord(self, dim: int, direction=ORIGIN):
    """Meant to generalize ``get_x``, ``get_y`` and ``get_z``"""
    return self.get_bounding_box_point(direction)[dim]

def get_x(self, direction=ORIGIN) -> np.float64:
    """Returns x coordinate of the center of the :class:`~.OpenGLMobject` as ``float``"""
    return self.get_coord(0, direction)

def get_y(self, direction=ORIGIN) -> np.float64:
    """Returns y coordinate of the center of the :class:`~.OpenGLMobject` as ``float``"""
    return self.get_coord(1, direction)

def get_z(self, direction=ORIGIN) -> np.float64:
    """Returns z coordinate of the center of the :class:`~.OpenGLMobject` as ``float``"""
    return self.get_coord(2, direction)

def get_start(self):
    """Returns the point, where the stroke that surrounds the :class:`~.OpenGLMobject` starts."""
    self.throw_error_if_no_points()
    return np.array(self.points[0])

def get_end(self):
    """Returns the point, where the stroke that surrounds the :class:`~.OpenGLMobject` ends."""
    self.throw_error_if_no_points()
    return np.array(self.points[-1])

def get_start_and_end(self):
    """Returns starting and ending point of a stroke as a ``tuple``."""
    return self.get_start(), self.get_end()

def point_from_proportion(self, alpha):
    points = self.points
    i, subalpha = integer_interpolate(0, len(points) - 1, alpha)
    return interpolate(points[i], points[i + 1], subalpha)

def pfp(self, alpha):
    """Abbreviation for point_from_proportion"""
    return self.point_from_proportion(alpha)

def get_pieces(self, n_pieces):
    template = self.copy()
    template.subobjects = []
    alphas = np.linspace(0, 1, n_pieces + 1)
    return OpenGLGroup(
        *(
            template.copy().pointwiseBecomePartial(self, a1, a2)
            for a1, a2 in zip(alphas[:-1], alphas[1:])
        )
    )

def get_z_index_reference_point(self):
    # TODO, better place to define default z_index_group?
    z_index_group = getattr(self, "z_index_group", self)
    return z_index_group.get_center()

```

```

# Match other mobject properties

def match_color(self, mobject: OpenGLMobject):
    """Match the color with the color of another :class:`~.OpenGLMobject`."""
    return self.set_color(mobject.get_color())

def match_dim_size(self, mobject: OpenGLMobject, dim, **kwargs):
    """Match the specified dimension with the dimension of another :class:`~.OpenGLMobject`."""
    return self.rescale_to_fit(mobject.length_over_dim(dim), dim, **kwargs)

def match_width(self, mobject: OpenGLMobject, **kwargs):
    """Match the width with the width of another :class:`~.OpenGLMobject`."""
    return self.match_dim_size(mobject, 0, **kwargs)

def match_height(self, mobject: OpenGLMobject, **kwargs):
    """Match the height with the height of another :class:`~.OpenGLMobject`."""
    return self.match_dim_size(mobject, 1, **kwargs)

def match_depth(self, mobject: OpenGLMobject, **kwargs):
    """Match the depth with the depth of another :class:`~.OpenGLMobject`."""
    return self.match_dim_size(mobject, 2, **kwargs)

def match_coord(self, mobject: OpenGLMobject, dim, direction=ORIGIN):
    """Match the coordinates with the coordinates of another :class:`~.OpenGLMobject`."""
    return self.set_coord(
        mobject.get_coord(dim, direction),
        dim=dim,
        direction=direction,
    )

def match_x(self, mobject, direction=ORIGIN):
    """Match x coord. to the x coord. of another :class:`~.OpenGLMobject`."""
    return self.match_coord(mobject, 0, direction)

def match_y(self, mobject, direction=ORIGIN):
    """Match y coord. to the x coord. of another :class:`~.OpenGLMobject`."""
    return self.match_coord(mobject, 1, direction)

def match_z(self, mobject, direction=ORIGIN):
    """Match z coord. to the x coord. of another :class:`~.OpenGLMobject`."""
    return self.match_coord(mobject, 2, direction)

def align_to(
    self,
    mobject_or_point: OpenGLMobject | Sequence[float],
    direction=ORIGIN,
) :
    """
    Examples:
    mob1.align_to(mob2, UP) moves mob1 vertically so that its
    top edge lines up with mob2's top edge.

    mob1.align_to(mob2, alignment_vect = RIGHT) moves mob1
    horizontally so that it's center is directly above/below
    the center of mob2
    """
    if isinstance(mobject_or_point, OpenGLMobject):
        point = mobject_or_point.get_bounding_box_point(direction)
    else:
        point = mobject_or_point

    for dim in range(self.dim):
        if direction[dim] != 0:
            self.set_coord(point[dim], dim, direction)
    return self

def get_group_class(self):
    return OpenGLGroup

@staticmethod
def get_mobject_type_class():
    """Return the base class of this mobject type."""
    return OpenGLMobject

# Alignment

def align_data_and_family(self, mobject):
    self.align_family(mobject)
    self.align_data(mobject)

def align_data(self, mobject):
    # In case any data arrays get resized when aligned to shader data
    # self.refresh_shader_data()
    for mob1, mob2 in zip(self.get_family(), mobject.get_family()):
        # Separate out how points are treated so that subclasses
        # can handle that case differently if they choose
        mob1.align_points(mob2)
        for key in mob1.data.keys() & mob2.data.keys():
            if key == "points":
                continue
            arr1 = mob1.data[key]
            arr2 = mob2.data[key]
```

```

        if len(arr2) > len(arr1):
            mob1.data[key] = resize_preserving_order(arr1, len(arr2))
        elif len(arr1) > len(arr2):
            mob2.data[key] = resize_preserving_order(arr2, len(arr1))

def align_points(self, mobject):
    max_len = max(self.get_num_points(), mobject.get_num_points())
    for mob in (self, mobject):
        mob.resize_points(max_len, resize_func=resize_preserving_order)
    return self

def align_family(self, mobject):
    mob1 = self
    mob2 = mobject
    n1 = len(mob1)
    n2 = len(mob2)
    if n1 != n2:
        mob1.add_n_more_submobjects(max(0, n2 - n1))
        mob2.add_n_more_submobjects(max(0, n1 - n2))
    # Recurse
    for sm1, sm2 in zip(mob1.submobjects, mob2.submobjects):
        sm1.align_family(sm2)
    return self

def push_self_into_submobjects(self):
    copy = self.deepcopy()
    copy.submobjects = []
    self.resize_points(0)
    self.add(copy)
    return self

def add_n_more_submobjects(self, n):
    if n == 0:
        return self

    curr = len(self.submobjects)
    if curr == 0:
        # If empty, simply add n point mobjects
        null_mob = self.copy()
        null_mob.set_points([self.get_center()])
        self.submobjects = [null_mob.copy() for k in range(n)]
        return self
    target = curr + n
    repeat_indices = (np.arange(target) * curr) // target
    split_factors = [(repeat_indices == i).sum() for i in range(curr)]
    new_submobs = []
    for submob, sf in zip(self.submobjects, split_factors):
        new_submobs.append(submob)
        for _ in range(1, sf):
            new_submob = submob.copy()
            # If the submob is at all transparent, then
            # make the copy completely transparent
            if submob.get_opacity() < 1:
                new_submob.set_opacity(0)
            new_submobs.append(new_submob)
    self.submobjects = new_submobs
    return self

# Interpolate

def interpolate(self, mobject1, mobject2, alpha, path_func=straight_path()):
    """Turns this :class:`~OpenGLMobject` into an interpolation between ``mobject1`` and ``mobject2``.

Examples
-----
.. manim:: DotInterpolation
:save_last_frame:

    class DotInterpolation(Scene):
        def construct(self):
            dotR = Dot(color=DARK_GREY)
            dotR.shift(2 * RIGHT)
            dotL = Dot(color=WHITE)
            dotL.shift(2 * LEFT)

            dotMiddle = OpenGLMobject().interpolate(dotL, dotR, alpha=0.3)

            self.add(dotL, dotR, dotMiddle)

    for key in self.data:
        if key in self.locked_data_keys:
            continue
        if len(self.data[key]) == 0:
            continue
        if key not in mobject1.data or key not in mobject2.data:
            continue

        if key in ("points", "bounding_box"):
            func = path_func
        else:
            func = interpolate

```

```

        self.data[key][:] = func(mobject1.data[key], mobject2.data[key], alpha)

    for key in self.uniforms:
        if key != "fixed_orientation_center":
            self.uniforms[key] = interpolate(
                mobject1.uniforms[key],
                mobject2.uniforms[key],
                alpha,
            )
        else:
            self.uniforms["fixed_orientation_center"] = tuple(
                interpolate(
                    np.array(mobject1.uniforms["fixed_orientation_center"]),
                    np.array(mobject2.uniforms["fixed_orientation_center"]),
                    alpha,
                )
            )
    )
return self

def pointwise_become_partial(self, mobject, a, b):
    """
    Set points in such a way as to become only
    part of mobject.
    Inputs 0 <= a < b <= 1 determine what portion
    of mobject to become.
    """
    pass # To implement in subclass

def become(
    self,
    mobject: OpenGLObject,
    match_height: bool = False,
    match_width: bool = False,
    match_depth: bool = False,
    match_center: bool = False,
    stretch: bool = False,
) :
    """
    Edit all data and submobjects to be identical
    to another :class:`~.OpenGLObject`

    .. note::

        If both match_height and match_width are ``True`` then the transformed :class:`~.OpenGLObject`
        will match the height first and then the width

    Parameters
    -----
    match_height
        If ``True``, then the transformed :class:`~.OpenGLObject` will match the height of the original
    match_width
        If ``True``, then the transformed :class:`~.OpenGLObject` will match the width of the original
    match_depth
        If ``True``, then the transformed :class:`~.OpenGLObject` will match the depth of the original
    match_center
        If ``True``, then the transformed :class:`~.OpenGLObject` will match the center of the original
    stretch
        If ``True``, then the transformed :class:`~.OpenGLObject` will stretch to fit the proportions of the original

    Examples
    -----
    .. manim:: BecomeScene

        class BecomeScene(Scene):
            def construct(self):
                circ = Circle(fill_color=RED, fill_opacity=0.8)
                square = Square(fill_color=BLUE, fill_opacity=0.2)
                self.add(circ)
                self.wait(0.5)
                circ.become(square)
                self.wait(0.5)

        if stretch:
            mobject.stretch_to_fit_height(self.height)
            mobject.stretch_to_fit_width(self.width)
            mobject.stretch_to_fit_depth(self.depth)
        else:
            if match_height:
                mobject.match_height(self)
            if match_width:
                mobject.match_width(self)
            if match_depth:
                mobject.match_depth(self)

        if match_center:
            mobject.move_to(self.get_center())

        self.align_family(mobject)
        for sm1, sm2 in zip(self.get_family(), mobject.get_family()):
            sm1.set_data(sm2.data)
            sm1.set_uniforms(sm2.uniforms)
        self.refresh_bounding_box(recurse_down=True)
    
```

```

    return self

# Locking data

def lock_data(self, keys):
    """
    To speed up some animations, particularly transformations,
    it can be handy to acknowledge which pieces of data
    won't change during the animation so that calls to
    interpolate can skip this, and so that it's not
    read into the shader_wrapper objects needlessly
    """
    if self.has_updaters:
        return
    # Be sure shader data has most up to date information
    self.refresh_shader_data()
    self.locked_data_keys = set(keys)

def lock_matching_data(self, mobject1, mobject2):
    for sm, sm1, sm2 in zip(
        self.get_family(),
        mobject1.get_family(),
        mobject2.get_family(),
    ):
        keys = sm.data.keys() & sm1.data.keys() & sm2.data.keys()
        sm.lock_data(
            list(
                filter(
                    lambda key: np.all(sm1.data[key] == sm2.data[key]),
                    keys,
                ),
            ),
        )
    return self

def unlock_data(self):
    for mob in self.get_family():
        mob.locked_data_keys = set()

# Operations touching shader uniforms

@affects_shader_info_id
def fix_in_frame(self):
    self.is_fixed_in_frame = 1.0
    return self

@affects_shader_info_id
def fix_orientation(self):
    self.is_fixed_orientation = 1.0
    self.fixed_orientation_center = tuple(self.get_center())
    self.depth_test = True
    return self

@affects_shader_info_id
def unfix_from_frame(self):
    self.is_fixed_in_frame = 0.0
    return self

@affects_shader_info_id
def unfix_orientation(self):
    self.is_fixed_orientation = 0.0
    self.fixed_orientation_center = (0, 0, 0)
    self.depth_test = False
    return self

@affects_shader_info_id
def apply_depth_test(self):
    self.depth_test = True
    return self

@affects_shader_info_id
def deactivate_depth_test(self):
    self.depth_test = False
    return self

# Shader code manipulation

def replace_shader_code(self, old, new):
    # TODO, will this work with VMobject structure, given
    # that it does not simpler return shader_wrappers of
    # family?
    for wrapper in self.get_shader_wrapper_list():
        wrapper.replace_code(old, new)
    return self

def set_color_by_code(self, glsl_code):
    """
    Takes a snippet of code and inserts it into a
    context which has the following variables:
    vec4 color, vec3 point, vec3 unit_normal.
    The code should change the color variable
    """
    self.replace_shader_code("////// INSERT COLOR FUNCTION HERE /////", glsl_code)

```

```

    return self

def set_color_by_xyz_func(
    self,
    glsl_snippet,
    min_value=-5.0,
    max_value=5.0,
    colormap="viridis",
):
    """
    Pass in a glsl expression in terms of x, y and z which returns
    a float.
    """
    # TODO, add a version of this which changes the point data instead
    # of the shader code
    for char in "xyz":
        glsl_snippet = glsl_snippet.replace(char, "point." + char)
    rgb_list = get_colormap_list(colormap)
    self.set_color_by_code(
        "color.rgb = float_to_color({}, {}, {}, {});".format(
            glsl_snippet,
            float(min_value),
            float(max_value),
            get_colormap_code(rgb_list),
        ),
    )
return self

# For shader data

def refresh_shader_wrapper_id(self):
    self.get_shader_wrapper().refresh_id()
    return self

def get_shader_wrapper(self):
    from manim.renderer.shader_wrapper import ShaderWrapper

    # if hasattr(self, "__shader_wrapper"):
    #     return self.__shader_wrapper

    self.shader_wrapper = ShaderWrapper(
        vert_data=self.get_shader_data(),
        vert_indices=self.get_shader_vert_indices(),
        uniforms=self.get_shader_uniforms(),
        depth_test=self.depth_test,
        texture_paths=self.texture_paths,
        render_primitive=self.render_primitive,
        shader_folder=self.__class__.shader_folder,
    )
    return self.shader_wrapper

def get_shader_wrapper_list(self):
    shader_wrappers = it.chain(
        [self.get_shader_wrapper()],
        *(sm.get_shader_wrapper_list() for sm in self.subobjects),
    )
    batches = batch_by_property(shader_wrappers, lambda sw: sw.get_id())

    result = []
    for wrapper_group, _ in batches:
        shader_wrapper = wrapper_group[0]
        if not shader_wrapper.is_valid():
            continue
        shader_wrapper.combine_with(*wrapper_group[1:])
        if len(shader_wrapper.vert_data) > 0:
            result.append(shader_wrapper)
    return result

def check_data_alignment(self, array, data_key):
    # Makes sure that self.data[key] can be broadcast into
    # the given array, meaning its length has to be either 1
    # or the length of the array
    d_len = len(self.data[data_key])
    if d_len != 1 and d_len != len(array):
        self.data[data_key] = resize_with_interpolation(
            self.data[data_key],
            len(array),
        )
    return self

def get_resized_shader_data_array(self, length):
    # If possible, try to populate an existing array, rather
    # than recreating it each frame
    points = self.points
    shader_data = np.zeros(len(points), dtype=self.shader_dtype)
    return shader_data

def read_data_to_shader(self, shader_data, shader_data_key, data_key):
    if data_key in self.locked_data_keys:
        return
    self.check_data_alignment(shader_data, data_key)
    shader_data[shader_data_key] = self.data[data_key]

```

```

def get_shader_data(self):
    shader_data = self.get_resized_shader_data_array(self.get_num_points())
    self.read_data_to_shader(shader_data, "point", "points")
    return shader_data

def refresh_shader_data(self):
    self.get_shader_data()

def get_shader_uniforms(self):
    return self.uniforms

def get_shader_vert_indices(self):
    return self.shader_indices

@property
def submobjects(self):
    return self._submobjects if hasattr(self, "_submobjects") else []

@submobjects.setter
def submobjects(self, submobject_list):
    self.remove(*self.submobjects)
    self.add(*submobject_list)

# Errors

def throw_error_if_no_points(self):
    if not self.has_points():
        message = (
            "Cannot call OpenGLObject.{} ".format(self.__class__.__name__)
            + "for a OpenGLObject with no points"
        )
        caller_name = sys._getframe(1).f_code.co_name
        raise Exception(message.format(caller_name))

class OpenGLGroup(OpenGLObject):
    def __init__(self, *mobjects, **kwargs):
        if not all(isinstance(m, OpenGLObject) for m in mobjects):
            raise Exception("All submobjects must be of type OpenGLObject")
        super().__init__(**kwargs)
        self.add(*mobjects)

class OpenGLPoint(OpenGLObject):
    def __init__(self, location=ORIGIN, artificial_width=1e-6, artificial_height=1e-6, **kwargs):
        self.artificial_width = artificial_width
        self.artificial_height = artificial_height
        super().__init__(**kwargs)
        self.set_location(location)

    def get_width(self):
        return self.artificial_width

    def get_height(self):
        return self.artificial_height

    def get_location(self):
        return self.points[0].copy()

    def get_bounding_box_point(self, *args, **kwargs):
        return self.get_location()

    def set_location(self, new_loc):
        self.set_points(np.array(new_loc, ndmin=2, dtype=float))

class AnimationBuilder:
    def __init__(self, mobject):
        self.mobject = mobject
        self.mobject.generate_target()

        self.overridden_animation = None
        self.is_chaining = False
        self.methods = []

        # Whether animation args can be passed
        self.cannot_pass_args = False
        self.anim_args = {}

    def __call__(self, **kwargs):
        if self.cannot_pass_args:
            raise ValueError(
                "Animation arguments must be passed before accessing methods and can only be passed once",
            )

        self.anim_args = kwargs
        self.cannot_pass_args = True

    def __getattr__(self, method_name):
        method = getattr(self.mobject.target, method_name)

```

```

has_overridden_animation = hasattr(method, "_override_animate")

if (self.is_chaining and has_overridden_animation) or self.overridden_animation:
    raise NotImplementedError(
        "Method chaining is currently not supported for "
        "overridden animations",
    )

def update_target(*method_args, **method_kwargs):
    if has_overridden_animation:
        self.overridden_animation = method._override_animate(
            self.mobject,
            *method_args,
            anim_args=self.anim_args,
            **method_kwargs,
        )
    else:
        self.methods.append([method, method_args, method_kwargs])
        method(*method_args, **method_kwargs)
    return self

self.is_chaining = True
self.cannot_pass_args = True

return update_target

def build(self):
    from manim.animation.transform import _MethodAnimation

    if self.overridden_animation:
        anim = self.overridden_animation
    else:
        anim = _MethodAnimation(self.mobject, self.methods)

    for attr, value in self.anim_args.items():
        setattr(anim, attr, value)

    return anim

def override_animate(method):
    """Decorator for overriding method animations.

    This allows to specify a method (returning an :class:`~.Animation`)
    which is called when the decorated method is used with the ``.animate`` syntax
    for animating the application of a method.

    .. seealso::

        :attr:`OpenGLMobject.animate`

    .. note::

        Overridden methods cannot be combined with normal or other overridden
        methods using method chaining with the ``.animate`` syntax.

    Examples
    -------

    .. manim:: AnimationOverrideExample

        class CircleWithContent(VGroup):
            def __init__(self, content):
                super().__init__()
                self.circle = Circle()
                self.content = content
                self.add(self.circle, content)
                content.move_to(self.circle.get_center())

            def clear_content(self):
                self.remove(self.content)
                self.content = None

            @override_animate(clear_content)
            def _clear_content_animation(self, anim_args=None):
                if anim_args is None:
                    anim_args = {}
                anim = Uncreate(self.content, **anim_args)
                self.clear_content()
                return anim

        class AnimationOverrideExample(Scene):
            def construct(self):
                t = Text("hello!")
                my_mobject = CircleWithContent(t)
                self.play(Create(my_mobject))
                self.play(my_mobject.animate.clear_content())
                self.wait()

    .. def decorator(animation_method):

```

```
method._override_animate = animation_method
return animation_method

return decorator
```

vector_field.py

```
"""Mobjects representing vector fields."""

from __future__ import annotations

__all__ = [
    "VectorField",
    "ArrowVectorField",
    "StreamLines",
]

import itertools as it
import random
from math import ceil, floor
from typing import Callable, Iterable, Sequence

import numpy as np
from colour import Color
from PIL import Image

from manim.animation.updaters import UpdateFromAlphaFunc
from manim.mobject.geometry.line import Vector
from manim.mobject.graphing.coordinate_systems import CoordinateSystem

from .. import config
from ..animation.composition import AnimationGroup, Succession
from ..animation.creation import Create
from ..animation.indication import ShowPassingFlash
from ..constants import OUT, RIGHT, UP, RendererType
from ..mobject.mobject import Mobject
from ..mobject.types.vectorized_mobject import VGroup
from ..mobject.utils import get_vectorized_mobject_class
from ..utils.bezier import interpolate, inverse_interpolate
from ..utils.color import BLUE_E, GREEN, RED, YELLOW, color_to_rgb, rgb_to_color
from ..utils.rate_functions import ease_out_sine, linear
from ..utils.simple_functions import sigmoid

DEFAULT_SCALAR_FIELD_COLORS: list = [BLUE_E, GREEN, YELLOW, RED]

class VectorField(VGroup):
    """A vector field.

    Vector fields are based on a function defining a vector at every position.
    This class does by default not include any visible elements but provides
    methods to move other :class:`~.Mobject`'s along the vector field.

    Parameters
    -----
    func
        The function defining the rate of change at every position of the `VectorField`.
    color
        The color of the vector field. If set, position-specific coloring is disabled.
    color_scheme
        A function mapping a vector to a single value. This value gives the position in the color gradient defined using `min_color_scheme_value`, `max_color_scheme_value` and `colors`
    min_color_scheme_value
        The value of the color_scheme function to be mapped to the first color in `colors`. Lower values also result in the first color of the gradient.
    max_color_scheme_value
        The value of the color_scheme function to be mapped to the last color in `colors`. Higher values also result in the last color of the gradient.
    colors
        The colors defining the color gradient of the vector field.
    kwargs
        Additional arguments to be passed to the :class:`~.VGroup` constructor
    """

    def __init__(self,
                 func: Callable[[np.ndarray], np.ndarray],
                 color: Color | None = None,
                 color_scheme: Callable[[np.ndarray], float] | None = None,
                 min_color_scheme_value: float = 0,
                 max_color_scheme_value: float = 2,
                 colors: Sequence[Color] = DEFAULT_SCALAR_FIELD_COLORS,
                 **kwargs,
                 ):
        super().__init__(**kwargs)
        self.func = func
        if color is None:
            self.single_color = False
        if color_scheme is None:
            def color_scheme(p):
                return np.linalg.norm(p)

        self.color_scheme = color_scheme # TODO maybe other default for direction?
        self.rgb = np.array(list(map(color_to_rgb, colors)))

        def pos_to_rgb(pos: np.ndarray) -> tuple[float, float, float, float]:
            vec = self.func(pos)
            color_value = np.clip(
                self.color_scheme(vec),
                min_color_scheme_value,
                max_color_scheme_value,
            )
            alpha = inverse_interpolate(
                min_color_scheme_value,
                max_color_scheme_value,
                color_value,
            )
            alpha *= len(self.rgb) - 1
            c1 = self.rgb[int(alpha)]
            c2 = self.rgb[min(int(alpha + 1), len(self.rgb) - 1)]
            alpha %= 1
            return interpolate(c1, c2, alpha)

        self.pos_to_rgb = pos_to_rgb
        self.pos_to_color = lambda pos: rgb_to_color(self.pos_to_rgb(pos))
        else:
            self.single_color = True
            self.color = color
            self.submob_movement_updater = None

    @staticmethod
    def shift_func(
        func: Callable[[np.ndarray], np.ndarray],
        shift_vector: np.ndarray,
    ) -> Callable[[np.ndarray], np.ndarray]:
        """Shift a vector field function.

        Parameters
        -----
        func
            The function defining a vector field.
        shift_vector
            The shift to be applied to the vector field.

        Returns
        -----
        Callable[[np.ndarray], np.ndarray]
            The shifted vector field function.
        """

        return lambda p: func(p - shift_vector)
```

```

@staticmethod
def scale_func(
    func: Callable[[np.ndarray], np.ndarray],
    scalar: float,
) -> Callable[[np.ndarray], np.ndarray]:
    """Scale a vector field function.

    Parameters
    -----
    func
        The function defining a vector field.
    scalar
        The scalar to be applied to the vector field.

    Examples
    -----
    .. manim:: ScaleVectorFieldFunction

        class ScaleVectorFieldFunction(Scene):
            def construct(self):
                func = lambda pos: np.sin(pos[1]) * RIGHT + np.cos(pos[0]) * UP
                vector_field = ArrowVectorField(func)
                self.add(vector_field)
                self.wait()

                func = VectorField.scale_func(func, 0.5)
                self.play(vector_field.animate.become(ArrowVectorField(func)))
                self.wait()

    Returns
    -----
    Callable[[np.ndarray], np.ndarray]
        The scaled vector field function.

    """
    return lambda p: func(p * scalar)

def fit_to_coordinate_system(self, coordinate_system: CoordinateSystem):
    """Scale the vector field to fit a coordinate system.

    This method is useful when the vector field is defined in a coordinate system
    different from the one used to display the vector field.

    This method can only be used once because it transforms the origin of each vector.

    Parameters
    -----
    coordinate_system
        The coordinate system to fit the vector field to.

    """
    self.apply_function(lambda pos: coordinate_system.coords_to_point(*pos))

def nudge(
    self,
    mob: Mobject,
    dt: float = 1,
    substeps: int = 1,
    pointwise: bool = False,
) -> VectorField:
    """Nudge a :class:`~.Mobject` along the vector field.

    Parameters
    -----
    mob
        The mobject to move along the vector field
    dt
        A scalar to the amount the mobject is moved along the vector field.
        The actual distance is based on the magnitude of the vector field.
    substeps
        The amount of steps the whole nudge is divided into. Higher values
        give more accurate approximations.
    pointwise
        Whether to move the mobject along the vector field. If `False` the
        vector field takes effect on the center of the given
        :class:`~.Mobject`. If `True` the vector field takes effect on the
        points of the individual points of the :class:`~.Mobject`,
        potentially distorting it.

    Returns
    -----
    VectorField
        This vector field.

    Examples
    -----
    .. manim:: Nudging

        class Nudging(Scene):
            def construct(self):
                func = lambda pos: np.sin(pos[1] / 2) * RIGHT + np.cos(pos[0] / 2) * UP
                vector_field = ArrowVectorField(
                    func, x_range=[-7, 7, 1], y_range=[-4, 4, 1], length_func=lambda x: x / 2
                )
                self.add(vector_field)
                circle = Circle(radius=2).shift(LEFT)
                self.add(circle.copy().set_color(GRAY))
                dot = Dot().move_to(circle)

                vector_field.nudge(circle, -2, 60, True)
                vector_field.nudge(dot, -2, 60)

                circle.add_updater(vector_field.get_nudge_updater(pointwise=True))
                dot.add_updater(vector_field.get_nudge_updater())
                self.add(circle, dot)
                self.wait(6)

    """

    def runge_kutta(self, p: Sequence[float], step_size: float) -> float:
        """Returns the change in position of a point along a vector field.

        Parameters
        -----
        p
            The position of each point being moved along the vector field.
        step_size
            A scalar that is used to determine how much a point is shifted in a single step.

        Returns
        -----
        float
            How much the point is shifted.

        """
        k_1 = self.func(p)
        k_2 = self.func(p + step_size * (k_1 * 0.5))
        k_3 = self.func(p + step_size * (k_2 * 0.5))
        k_4 = self.func(p + step_size * k_3)
        return step_size / 6.0 * (k_1 + 2.0 * k_2 + 2.0 * k_3 + k_4)

    step_size = dt / substeps
    for _ in range(substeps):
        if pointwise:
            mob.apply_function(lambda p: p + runge_kutta(self, p, step_size))
        else:
            mob.shift(runge_kutta(self, mob.get_center(), step_size))
    return self

```

```

def nudge_submobjects(
    self,
    dt: float = 1,
    substeps: int = 1,
    pointwise: bool = False,
) -> VectorField:
    """Apply a nudge along the vector field to all submobjects.

    Parameters
    -----
    dt
        A scalar to the amount the mob object is moved along the vector field.
        The actual distance is based on the magnitude of the vector field.
    substeps
        The amount of steps the whole nudge is divided into. Higher values
        give more accurate approximations.
    pointwise
        Whether to move the mob object along the vector field. See :meth:`nudge` for details.

    Returns
    -----
    VectorField
        This vector field.

    """
    for mob in self.submobjects:
        self.nudge(mob, dt, substeps, pointwise)
    return self

def get_nudge_updater(
    self,
    speed: float = 1,
    pointwise: bool = False,
) -> Callable[[Mobject, float], Mobject]:
    """Get an update function to move a :class:`~.Mobject` along the vector field.

    When used with :meth:`~.Mobject.add_updater`, the mob object will move along the vector field, where its speed is determined by the magnitude of the vector field.

    Parameters
    -----
    speed
        At `speed=1` the distance a mob object moves per second is equal to the magnitude of the vector field along its path. The speed value scales the speed of such a mob object.
    pointwise
        Whether to move the mob object along the vector field. See :meth:`nudge` for details.

    Returns
    -----
    Callable[[Mobject, float], Mobject]
        The update function.

    """
    return lambda mob, dt: self.nudge(mob, dt * speed, pointwise=pointwise)

def start_submobject_movement(
    self,
    speed: float = 1,
    pointwise: bool = False,
) -> VectorField:
    """Start continuously moving all submobjects along the vector field.

    Calling this method multiple times will result in removing the previous updater created by this method.

    Parameters
    -----
    speed
        The speed at which to move the submobjects. See :meth:`get_nudge_updater` for details.
    pointwise
        Whether to move the mob object along the vector field. See :meth:`nudge` for details.

    Returns
    -----
    VectorField
        This vector field.

    """
    self.stop_submobject_movement()
    self.submob_movement_updater = lambda mob, dt: mob.nudge_submobjects(
        dt * speed,
        pointwise=pointwise,
    )
    self.add_updater(self.submob_movement_updater)
    return self

def stop_submobject_movement(self) -> VectorField:
    """Stops the continuous movement started using :meth:`start_submobject_movement`.

    Returns
    -----
    VectorField
        This vector field.

    """
    self.remove_updater(self.submob_movement_updater)
    self.submob_movement_updater = None
    return self

def get_colored_background_image(self, sampling_rate: int = 5) -> Image.Image:
    """Generate an image that displays the vector field.

    The color at each position is calculated by passing the position through a
    series of steps:
    Calculate the vector field function at that position, map that vector to a
    single value using `self.color_scheme` and finally generate a color from
    that value using the color gradient.

    Parameters
    -----
    sampling_rate
        The stepsize at which pixels get included in the image. Lower values give
        more accurate results, but may take a long time to compute.

    Returns
    -----
    Image.Image
        The vector field image.

    """
    if self.single_color:
        raise ValueError(
            "There is no point in generating an image if the vector field uses a single color."
        )
    ph = int(config["pixel_height"] / sampling_rate)
    pw = int(config["pixel_width"] / sampling_rate)
    fw = config["frame_width"]
    fh = config["frame_height"]
    points_array = np.zeros((ph, pw, 3))
    x_array = np.linspace(-fw / 2, fw / 2, pw)
    y_array = np.linspace(fh / 2, -fh / 2, ph)
    x_array = x_array.reshape((1, len(x_array)))
    y_array = y_array.reshape((len(y_array), 1))
    x_array = x_array.repeat(ph, axis=0)
    y_array.repeat(pw, axis=1) # TODO why not y_array = y_array.repeat(...)?
    points_array[:, :, 0] = x_array
    points_array[:, :, 1] = y_array
    rgbs = np.apply_along_axis(self.pos_to_rgb, 2, points_array)
    return Image.fromarray((rgbs * 255).astype("uint8"))

def get_vectorized_rgbs_gradient_function(
    self,
    start: float,
    end: float,
)

```

```

colors: Iterable,
):
"""
Generates a gradient of rgbs as a numpy array

Parameters
-----
start
    start value used for inverse interpolation at :func:`~.inverse_interpolate`
end
    end value used for inverse interpolation at :func:`~.inverse_interpolate`
colors
    list of colors to generate the gradient

Returns
-----
function to generate the gradients as numpy arrays representing rgba values
"""

rgbs = np.array([color_to_rgb(c) for c in colors])

def func(values, opacity=1):
    alphas = inverse_interpolate(start, end, np.array(values))
    alphas = np.clip(alphas, 0, 1)
    scaled_alphas = alphas * (len(rgbs) - 1)
    indices = scaled_alphas.astype(int)
    next_indices = np.clip(indices + 1, 0, len(rgbs) - 1)
    inter_alphas = scaled_alphas % 1
    inter_alphas = inter_alphas.repeat(3).reshape((len(indices), 3))
    result = interpolate(rgbs[indices], rgbs[next_indices], inter_alphas)
    result = np.concatenate(
        (result, np.full(len(result), 1, 1), opacity)),
    axis=1,
)
    return result

return func

class ArrowVectorField(VectorField):
    """A :class:`VectorField` represented by a set of change vectors.

    Vector fields are always based on a function defining the :class:`~.Vector` at every position.
    The values of this functions is displayed as a grid of vectors.
    By default the color of each vector is determined by it's magnitude.
    Other color schemes can be used however.

    Parameters
    -----
    func
        The function defining the rate of change at every position of the vector field.
    color
        The color of the vector field. If set, position-specific coloring is disabled.
    color_scheme
        A function mapping a vector to a single value. This value gives the position in the color gradient defined using `min_color_scheme_value`, `max_color_scheme_value` and `colors`.
    min_color_scheme_value
        The value of the color_scheme function to be mapped to the first color in `colors`. Lower values also result in the first color of the gradient.
    max_color_scheme_value
        The value of the color_scheme function to be mapped to the last color in `colors`. Higher values also result in the last color of the gradient.
    colors
        The colors defining the color gradient of the vector field.
    x_range
        A sequence of x_min, x_max, delta_x
    y_range
        A sequence of y_min, y_max, delta_y
    z_range
        A sequence of z_min, z_max, delta_z
    three_dimensions
        Enables three dimensions. Default set to False, automatically turns True if z_range is not None.
    length_func
        The function determining the displayed size of the vectors. The actual size of the vector is passed, the returned value will be used as display size for the vector. By default this is used to cap the displayed size of vectors to reduce the clutter.
    opacity
        The opacity of the arrows.
    vector_config
        Additional arguments to be passed to the :class:`~.Vector` constructor
    kwargs
        Additional arguments to be passed to the :class:`~.VGroup` constructor

    Examples
    -----
    .. manim:: BasicUsage
        :save_last_frame:

        class BasicUsage(Scene):
            def construct(self):
                func = lambda pos: ((pos[0] * UR + pos[1] * LEFT) - pos) / 3
                self.add(ArrowVectorField(func))

    .. manim:: SizingAndSpacing

        class SizingAndSpacing(Scene):
            def construct(self):
                func = lambda pos: np.sin(pos[0] / 2) * UR + np.cos(pos[1] / 2) * LEFT
                vf = ArrowVectorField(func, x_range=[-7, 7, 1])
                self.add(vf)
                self.wait()

                length_func = lambda x: x / 3
                vf2 = ArrowVectorField(func, x_range=[-7, 7, 1], length_func=length_func)
                self.play(vf.animate.become(vf2))
                self.wait()

    .. manim:: Coloring
        :save_last_frame:

        class Coloring(Scene):
            def construct(self):
                func = lambda pos: pos - LEFT * 5
                colors = [RED, YELLOW, BLUE, DARK_GRAY]
                min_radius = Circle(radius=2, color=colors[0]).shift(LEFT * 5)
                max_radius = Circle(radius=10, color=colors[-1]).shift(LEFT * 5)
                vf = ArrowVectorField(
                    func, min_color_scheme_value=2, max_color_scheme_value=10, colors=colors
                )
                self.add(vf, min_radius, max_radius)

    """
    def __init__(self,
                 func: Callable[[np.ndarray], np.ndarray],
                 color: Color | None = None,
                 color_scheme: Callable[[np.ndarray], float] | None = None,
                 min_color_scheme_value: float = 0,
                 max_color_scheme_value: float = 2,
                 colors: Sequence[Color] = DEFAULT_SCALAR_FIELD_COLORS,
                 # Determining Vector positions:
                 x_range: Sequence[float] = None,
                 y_range: Sequence[float] = None,
                 z_range: Sequence[float] = None,
                 three_dimensions: bool = False, # Automatically True if z_range is set
                 # Taken in actual norm, spits out displayed norm
                 length_func: Callable[[float], float] = lambda norm: 0.45 * sigmoid(norm),
                 opacity: float = 1.0,
                 vector_config: dict | None = None,
                 **kwargs,
    ):

```

```

):
    self.x_range = x_range or [
        floor(-config["frame_width"] / 2),
        ceil(config["frame_width"] / 2),
    ]
    self.y_range = y_range or [
        floor(-config["frame_height"] / 2),
        ceil(config["frame_height"] / 2),
    ]
    self.ranges = [self.x_range, self.y_range]

    if three_dimensions or z_range:
        self.z_range = z_range or self.y_range.copy()
        self.ranges += [self.z_range]
    else:
        self.ranges += [[0, 0]]

    for i in range(len(self.ranges)):
        if len(self.ranges[i]) == 2:
            self.ranges[i] += [0.5]
            self.ranges[i][1] += self.ranges[i][2]

    self.x_range, self.y_range, self.z_range = self.ranges

super().__init__(
    func,
    color,
    color_scheme,
    min_color_scheme_value,
    max_color_scheme_value,
    colors,
    **kwargs,
)

self.length_func = length_func
self.opacity = opacity
if vector_config is None:
    vector_config = {}
self.vector_config = vector_config
self.func = func

x_range = np.arange(*self.x_range)
y_range = np.arange(*self.y_range)
z_range = np.arange(*self.z_range)
self.add(
    *[
        self.get_vector(x * RIGHT + y * UP + z * OUT)
        for x, y, z in it.product(x_range, y_range, z_range)
    ]
)
self.set_opacity(self.opacity)

def get_vector(self, point: np.ndarray):
    """Creates a vector in the vector field.

    The created vector is based on the function of the vector field and is
    rooted in the given point. Color and length fit the specifications of
    this vector field.

    Parameters
    -----
    point
        The root point of the vector.

    """
    output = np.asarray(self.func(point))
    norm = np.linalg.norm(output)
    if norm != 0:
        output *= self.length_func(norm) / norm
    vect = Vector(output, **self.vector_config)
    vect.shift(point)
    if self.single_color:
        vect.set_color(self.color)
    else:
        vect.set_color(self.pos_to_color(point))
    return vect

class StreamLines(VectorField):
    """StreamLines represent the flow of a :class:`VectorField` using the trace of moving agents.

    Vector fields are always based on a function defining the vector at every position.
    The values of this functions is displayed by moving many agents along the vector field
    and showing their trace.

    Parameters
    -----
    func
        The function defining the rate of change at every position of the vector field.
    color
        The color of the vector field. If set, position-specific coloring is disabled.
    color_scheme
        A function mapping a vector to a single value. This value gives the position in the color gradient defined using 'min_color_scheme_value', 'max_color_scheme_value' and 'colors'
    min_color_scheme_value
        The value of the color_scheme function to be mapped to the first color in 'colors'. Lower values also result in the first color of the gradient.
    max_color_scheme_value
        The value of the color_scheme function to be mapped to the last color in 'colors'. Higher values also result in the last color of the gradient.
    colors
        The colors defining the color gradient of the vector field.
    x_range
        A sequence of x_min, x_max, delta_x
    y_range
        A sequence of y_min, y_max, delta_y
    z_range
        A sequence of z_min, z_max, delta_z
    three_dimensions
        Enables three_dimensions. Default set to False, automatically turns True if
        z_range is not None.
    noise_factor
        The amount by which the starting position of each agent is altered along each axis. Defaults to :code:`delta_y / 2` if not defined.
    n_repeats
        The number of agents generated at each starting point.
    dt
        The factor by which the distance an agent moves per step is stretched. Lower values result in a better approximation of the trajectories in the vector field.
    virtual_time
        The time the agents get to move in the vector field. Higher values therefore result in longer stream lines. However, this whole time gets simulated upon creation.
    max_anchors_per_line
        The maximum number of anchors per line. Lines with more anchors get reduced in complexity, not in length.
    padding
        The distance agents can move out of the generation area before being terminated.
    stroke_width
        The stroke width of the stream lines.
    opacity
        The opacity of the stream lines.

    Examples
    -----
    .. manim:: BasicUsage
       :save_last_frame:

    class BasicUsage(Scene):
        def construct(self):
            func = lambda pos: ((pos[0] * UR + pos[1] * LEFT) - pos) / 3
            self.add(StreamLines(func))

    .. manim:: SpawningAndFlowingArea
       :save_last_frame:

```

```

class SpawningAndFlowingArea(Scene):
    def construct(self):
        func = lambda pos: np.sin(pos[0]) * UR + np.cos(pos[1]) * LEFT + pos / 5
        stream_lines = Streamlines(
            func, x_range=[-3, 3, 0.2], y_range=[-2, 2, 0.2], padding=1
        )

        spawning_area = Rectangle(width=6, height=4)
        flowing_area = Rectangle(width=8, height=6)
        labels = [Tex("Spawning Area", Tex("Flowing Area").shift(DOWN * 2.5)]
        for lbl in labels:
            lbl.add_background_rectangle(opacity=0.6, buff=0.05)

        self.add(stream_lines, spawning_area, flowing_area, *labels)

    """
    def __init__(self,
                 func: Callable[[np.ndarray], np.ndarray],
                 color: Color | None = None,
                 color_scheme: Callable[[np.ndarray], float] | None = None,
                 min_color_scheme_value: float = 0,
                 max_color_scheme_value: float = 2,
                 colors: Sequence[Color] = DEFAULT_SCALAR_FIELD_COLORS,
                 # Determining stream line starting positions:
                 x_range: Sequence[float] = None,
                 y_range: Sequence[float] = None,
                 z_range: Sequence[float] = None,
                 three_dimensions: bool = False,
                 noise_factor: float | None = None,
                 n_repeats=1,
                 # Determining how lines are drawn
                 dt=0.05,
                 virtual_time=3,
                 max_anchors_per_line=100,
                 padding=3,
                 # Determining stream line appearance:
                 stroke_width=1,
                 opacity=1,
                 **kwargs,
                 ):
        self.x_range = x_range or [
            floor(-config["frame_width"] / 2),
            ceil(config["frame_width"] / 2),
        ]
        self.y_range = y_range or [
            floor(-config["frame_height"] / 2),
            ceil(config["frame_height"] / 2),
        ]
        self.ranges = [self.x_range, self.y_range]

        if three_dimensions or z_range:
            self.z_range = z_range or self.y_range.copy()
            self.ranges += [self.z_range]
        else:
            self.ranges += [[0, 0]]

        for i in range(len(self.ranges)):
            if len(self.ranges[i]) == 2:
                self.ranges[i] += [0.5]
                self.ranges[i][1] += self.ranges[i][2]

        self.x_range, self.y_range, self.z_range = self.ranges

        super().__init__(
            func,
            color,
            color_scheme,
            min_color_scheme_value,
            max_color_scheme_value,
            colors,
            **kwargs,
        )

        self.noise_factor = (
            noise_factor if noise_factor is not None else self.y_range[2] / 2
        )
        self.n_repeats = n_repeats
        self.virtual_time = virtual_time
        self.max_anchors_per_line = max_anchors_per_line
        self.padding = padding
        self.stroke_width = stroke_width

        half_noise = self.noise_factor / 2
        np.random.seed(0)
        start_points = np.array([
            (x - half_noise) * RIGHT
            + (y - half_noise) * UP
            + (z - half_noise) * OUT
            + self.noise_factor * np.random.random(3)
            for n in range(self.n_repeats)
            for x in np.arange(*self.x_range)
            for y in np.arange(*self.y_range)
            for z in np.arange(*self.z_range)
        ]),
    )

    def outside_box(p):
        return (
            p[0] < self.x_range[0] - self.padding
            or p[0] > self.x_range[1] + self.padding - self.x_range[2]
            or p[1] < self.y_range[0] - self.padding
            or p[1] > self.y_range[1] + self.padding - self.y_range[2]
            or p[2] < self.z_range[0] - self.padding
            or p[2] > self.z_range[1] + self.padding - self.z_range[2]
        )

    max_steps = ceil(virtual_time / dt) + 1
    if not self.single_color:
        self.background_img = self.get_colored_background_image()
        if config["renderer"] == RenderType.OPENGL:
            self.values_to_rgbs = self.get_vectorized_rgba_gradient_function(
                min_color_scheme_value,
                max_color_scheme_value,
                colors,
            )
        for point in start_points:
            points = [point]
            for i in range(max_steps):
                last_point = points[-1]
                new_point = last_point + dt * func(last_point)
                if outside_box(new_point):
                    break
                points.append(new_point)
            step = max_steps
            if not step:
                continue
            line = get_vectorized_mobject_class()()
            line.duration = step * dt
            step = max(1, int(len(points) / self.max_anchors_per_line))
            line.set_points_smoothly(points[:step])
            if self.single_color:
                line.set_stroke(self.color)
            else:
                if config.renderer == RenderType.OPENGL:
                    # scaled for compatibility with cairo
                    line.set_stroke(width=self.stroke_width / 4.0)

```

```

norms = np.array(
    [np.linalg.norm(self.func(point)) for point in line.points],
)
line.set_rgba_array_direct(
    self.values_to_rgbs(norms, opacity),
    name="stroke_rgba",
)
else:
    if np.any(self.z_range != np.array([0, 0.5, 0.5])):
        line.set_stroke(
            [self.pos_to_color(p) for p in line.get_anchors()],
        )
    else:
        line.color_using_background_image(self.background_img)
        line.set_stroke(width=self.stroke_width, opacity=opacity)
self.add(line)
self.stream_lines = [*self.submobjects]

def create(
    self,
    lag_ratio: float | None = None,
    run_time: Callable[[float], float] | None = None,
    **kwargs,
) -> AnimationGroup:
    """The creation animation of the stream lines.

    The stream lines appear in random order.

    Parameters
    -----
    lag_ratio
        The lag ratio of the animation.
        If undefined, it will be selected so that the total animation length is 1.5 times the run time of each stream line creation.
    run_time
        The run time of every single stream line creation. The runtime of the whole animation might be longer due to the `lag_ratio`.
        If undefined, the virtual time of the stream lines is used as run time.

    Returns
    -----
    :class:`~.AnimationGroup`
        The creation animation of the stream lines.

    Examples
    -----
    .. manim:: StreamLineCreation

    class StreamLineCreation(Scene):
        def construct(self):
            func = lambda pos: (pos[0] * UR + pos[1] * LEFT) - pos
            stream_lines = StreamLines(
                func,
                color=YELLOW,
                x_range=[-7, 7, 1],
                y_range=[-4, 4, 1],
                stroke_width=3,
                virtual_time=1, # use shorter lines
                max_anchors_per_line=5, # better performance with fewer anchors
            )
            self.play(stream_lines.create()) # uses virtual_time as run_time
            self.wait()

    """
    if run_time is None:
        run_time = self.virtual_time
    if lag_ratio is None:
        lag_ratio = run_time / 2 / len(self.submobjects)

    animations = [
        Create(line, run_time=run_time, **kwargs) for line in self.stream_lines
    ]
    random.shuffle(animations)
    return AnimationGroup(*animations, lag_ratio=lag_ratio)

def start_animation(
    self,
    warm_up: bool = True,
    flow_speed: float = 1,
    time_width: float = 0.3,
    rate_func: Callable[[float], float] = linear,
    line_animation_class: type[ShowPassingFlash] = ShowPassingFlash,
    **kwargs,
) -> None:
    """Animates the stream lines using an updater.

    The stream lines will continuously flow

    Parameters
    -----
    warm_up
        If 'True' the animation is initialized line by line. Otherwise it starts with all lines shown.
    flow_speed
        At flow_speed=1 the distance the flow moves per second is equal to the magnitude of the vector field along its path. The speed value scales the speed of this flow.
    time_width
        The proportion of the stream line shown while being animated
    rate_func
        The rate function of each stream line flashing
    line_animation_class
        The animation class being used

    Examples
    -----
    .. manim:: ContinuousMotion

    class ContinuousMotion(Scene):
        def construct(self):
            func = lambda pos: np.sin(pos[0] / 2) * UR + np.cos(pos[1] / 2) * LEFT
            stream_lines = StreamLines(func, stroke_width=3, max_anchors_per_line=30)
            self.add(stream_lines)
            stream_lines.start_animation(warm_up=False, flow_speed=1.5)
            self.wait(stream_lines.virtual_time / stream_lines.flow_speed)

    """
    for line in self.stream_lines:
        run_time = line.duration / flow_speed
        line.anim = line_animation_class(
            line,
            run_time=run_time,
            rate_func=rate_func,
            time_width=time_width,
            **kwargs,
        )
        line.anim.begin()
        line.time = random.random() * self.virtual_time
        if warm_up:
            line.time *= -1
        self.add(line.anim.mobject)

    def updater(mob, dt):
        for line in mob.stream_lines:
            line.time += dt * flow_speed
            if line.time >= self.virtual_time:
                line.time -= self.virtual_time
            line.anim.interpolate(np.clip(line.time / line.anim.run_time, 0, 1))

    self.add_updater(updater)
    self.flow_animation = updater

```

```

self.flow_speed = flow_speed
self.time_width = time_width

def end_animation(self) -> AnimationGroup:
    """End the stream line animation smoothly.

    Returns an animation resulting in fully displayed stream lines without a noticeable cut.

    Returns
    ------
    :class:`~.AnimationGroup`:
        The animation fading out the running stream animation.

    Raises
    ------
    ValueError
        if no stream line animation is running

    Examples
    ------
    .. manim:: EndAnimation

        class EndAnimation(Scene):
            def construct(self):
                func = lambda pos: np.sin(pos[0] / 2) * UR + np.cos(pos[1] / 2) * LEFT
                stream_lines = StreamLines(
                    func, stroke_width=3, max_anchors_per_line=5, virtual_time=1, color=BLUE
                )
                self.add(stream_lines)
                stream_lines.start_animation(warm_up=False, flow_speed=1.5, time_width=0.5)
                self.wait(1)
                self.play(stream_lines.end_animation())

    """
    if self.flow_animation is None:
        raise ValueError("You have to start the animation before fading it out.")

    def hide_and_wait(mob, alpha):
        if alpha == 0:
            mob.set_stroke(opacity=0)
        elif alpha == 1:
            mob.set_stroke(opacity=1)

    def finish_updater_cycle(line, alpha):
        line.time += dt * self.flow_speed
        line.anim.interpolate(min(line.time / line.anim.run_time, 1))
        if alpha == 1:
            self.remove(line.anim.mobject)
            line.anim.finish()

    max_run_time = self.virtual_time / self.flow_speed
    creation_rate_func = ease_out_sine
    creation_staring_speed = creation_rate_func(0.001) * 1000
    creation_run_time = (
        max_run_time / (1 + self.time_width) * creation_staring_speed
    )
    # creation_run_time is calculated so that the creation animation starts at the same speed
    # as the regular line flash animation but eases out.

    dt = 1 / config["frame_rate"]
    animations = []
    self.remove_updater(self.flow_animation)
    self.flow_animation = None

    for line in self.stream_lines:
        create = Create(
            line,
            run_time=creation_run_time,
            rate_func=creation_rate_func,
        )
        if line.time <= 0:
            animations.append(
                Succession(
                    UpdateFromAlphaFunc(
                        line,
                        hide_and_wait,
                        run_time=-line.time / self.flow_speed,
                    ),
                    create,
                ),
            )
            self.remove(line.anim.mobject)
            line.anim.finish()
        else:
            remaining_time = max_run_time - line.time / self.flow_speed
            animations.append(
                Succession(
                    UpdateFromAlphaFunc(
                        line,
                        finish_updater_cycle,
                        run_time=remaining_time,
                    ),
                    create,
                ),
            )
    return AnimationGroup(*animations)

# TODO: Variant of StreamLines that is able to respond to changes in the vector field function

```

vectorized_mobject.py

```
"""Mobjects that use vector graphics."""

from __future__ import annotations

__all__ = [
    "VMobject",
    "VGroup",
    "VDict",
    "VectorizedPoint",
    "CurvesAsSubmobjects",
    "DashedVMobject",
]

import itertools as it
import sys
import typing
from typing import Callable, Optional, Sequence, Union

import colour
import numpy as np
from PIL.Image import Image

from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl.opengl_vectorized_mobject import OpenGLVMobject
from manim.mobject.three_d.three_d_utils import (
    get_3d_vmob_gradient_start_and_end_points,
)

from ... import config
from ...constants import *
from ...mobject.mobject import Mobject
from ...utils.bezier import (
    bezier,
    get_smooth_handle_points,
    integer_interpolate,
    interpolate,
    partial_bezier_points,
    proportions_along_bezier_curve_for_point,
)
from ...utils.color import BLACK, WHITE, color_to_rgba
from ...utils.deprecation import deprecated
from ...utils.iterables import make_even, resize_array, stretch_array_to_length, tuplify
from ...utils.space_ops import rotate_vector, shoelace_direction

# TODO
# - Change cubic curve groups to have 4 points instead of 3
# - Change sub_path idea accordingly
# - No more mark_paths_closed, instead have the camera test
# if last point in close to first point
# - Think about length of self.points. Always 0 or 1 mod 4?
# That's kind of weird.

class VMobject(Mobject):
    """A vectorized mobject.

    Parameters
    -----
    background_stroke_color
        The purpose of background stroke is to have something
        that won't overlap fill, e.g. For text against some
        textured background.
    sheen_factor
        When a color c is set, there will be a second color
        computed based on interpolating c to WHITE by with
        sheen_factor, and the display will gradient to this
        secondary color in the direction of sheen_direction.
    close_new_points
        Indicates that it will not be displayed, but
        that it should count in parent mobject's path
    tolerance_for_point_equality
        This is within a pixel
    joint_type
        The line joint type used to connect the curve segments
        of this vectorized mobject. See :class:`.LineJointType`
        for options.
    """

    sheen_factor = 0.0

    def __init__(
        self,
        fill_color=None,
        fill_opacity=0.0,
        stroke_color=None,
        stroke_opacity=1.0,
        stroke_width=DEFAULT_STROKE_WIDTH,
        background_stroke_color=BLACK,
        background_stroke_opacity=1.0,
        background_stroke_width=0,
        sheen_factor=0.0,
        joint_type: LineJointType | None = None,
        sheen_direction=UL,
        close_new_points=False,
        pre_function_handle_to_anchor_scale_factor=0.01,
        make_smooth_after_applying_functions=False,
        background_image=None,
        shade_in_3d=False,
        # TODO, do we care about accounting for varying zoom levels?
        tolerance_for_point_equality=1e-6,
        n_points_per_cubic_curve=4,
        **kwargs,
    ):
        self.fill_opacity = fill_opacity
        self.stroke_opacity = stroke_opacity
        self.stroke_width = stroke_width
        self.background_stroke_color = background_stroke_color
        self.background_stroke_opacity = background_stroke_opacity
        self.background_stroke_width = background_stroke_width
        self.sheen_factor = sheen_factor
        if joint_type is None:
            joint_type = LineJointType.AUTO
        self.joint_type = joint_type
        self.sheen_direction = sheen_direction
        self.close_new_points = close_new_points
        self.pre_function_handle_to_anchor_scale_factor = (
```

```

    pre_function_handle_to_anchor_scale_factor
)
self.make_smooth_after_applying_functions = make_smooth_after_applying_functions
self.background_image = background_image
self.shade_in_3d = shade_in_3d
self.tolerance_for_point_equality = tolerance_for_point_equality
self.n_points_per_cubic_curve = n_points_per_cubic_curve
super().__init__(**kwargs)

if fill_color:
    self.fill_color = fill_color
if stroke_color:
    self.stroke_color = stroke_color

# OpenGL compatibility
@property
def n_points_per_curve(self):
    return self.n_points_per_cubic_curve

def get_group_class(self):
    return VGroup

@staticmethod
def get_mobject_type_class():
    return VMobject

# Colors
def init_colors(self, propagate_colors=True):
    self.set_fill(
        color=self.fill_color,
        opacity=self.fill_opacity,
        family=propagate_colors,
    )
    self.set_stroke(
        color=self.stroke_color,
        width=self.stroke_width,
        opacity=self.stroke_opacity,
        family=propagate_colors,
    )
    self.set_background_stroke(
        color=self.background_stroke_color,
        width=self.background_stroke_width,
        opacity=self.background_stroke_opacity,
        family=propagate_colors,
    )
    self.set_sheen(
        factor=self.sheen_factor,
        direction=self.sheen_direction,
        family=propagate_colors,
    )

    if not propagate_colors:
        for submobject in self.submobjects:
            submobject.init_colors(propagate_colors=False)

return self

def generate_rgbs_array(self, color, opacity):
    """
    First arg can be either a color, or a tuple/list of colors.
    Likewise, opacity can either be a float, or a tuple of floats.
    If self.sheen_factor is not zero, and only
    one color was passed in, a second slightly light color
    will automatically be added for the gradient
    """
    colors = [c if (c is not None) else BLACK for c in tuplify(color)]
    opacities = [o if (o is not None) else 0 for o in tuplify(opacity)]
    rgbs = np.array([
        [color_to_rgba(c, o) for c, o in zip(*make_even(colors, opacities))],
    ])

    sheen_factor = self.get_sheen_factor()
    if sheen_factor != 0 and len(rgbs) == 1:
        light_rgbs = np.array(rgbs)
        light_rgbs[:, :3] += sheen_factor
        np.clip(light_rgbs, 0, 1, out=light_rgbs)
        rgbs = np.append(rgbs, light_rgbs, axis=0)
    return rgbs

def update_rgbs_array(self, array_name, color=None, opacity=None):
    rgbs = self.generate_rgbs_array(color, opacity)
    if not hasattr(self, array_name):
        setattr(self, array_name, rgbs)
    return self

    # Match up current rgbs array with the newly calculated
    # one. 99% of the time they'll be the same.
    curr_rgbs = getattr(self, array_name)
    if len(curr_rgbs) < len(rgbs):
        curr_rgbs = stretch_array_to_length(curr_rgbs, len(rgbs))
        setattr(self, array_name, curr_rgbs)
    elif len(rgbs) < len(curr_rgbs):
        rgbs = stretch_array_to_length(rgbs, len(curr_rgbs))
    # Only update rgb if color was not None, and only
    # update alpha channel if opacity was passed in
    if color is not None:
        curr_rgbs[:, :3] = rgbs[:, :3]
    if opacity is not None:
        curr_rgbs[:, 3] = rgbs[:, 3]
    return self

def set_fill(
    self,
    color: str | None = None,
    opacity: float | None = None,
    family: bool = True,
):
    """
    Set the fill color and fill opacity of a :class:`VMobject`.

    Parameters
    -----
    color
        Fill color of the :class:`VMobject`.
    opacity
        Fill opacity of the :class:`VMobject`.
    family
        If ``True``, the fill color of all submobjects is also set.

    Returns
    -----
    :class:`VMobject`
        ``self``
    """

Examples

```

```

-----
.. manim:: SetFill
:save_last_frame:

class SetFill(Scene):
    def construct(self):
        square = Square().scale(2).set_fill(WHITE,1)
        circle1 = Circle().set_fill(GREEN,0.8)
        circle2 = Circle().set_fill(YELLOW) # No fill_opacity
        circle3 = Circle().set_fill(color = '#FF2135', opacity = 0.2)
        group = Group(circle1,circle2,circle3).arrange()
        self.add(square)
        self.add(group)

See Also
-----
:meth:`~.VMobject.set_style`  

"""

if family:
    for submobject in self.submobjects:
        submobject.set_fill(color, opacity, family)
    self.update_rgbs_array("fill_rgbs", color, opacity)
if opacity is not None:
    self.fill_opacity = opacity
return self

def set_stroke(
    self,
    color=None,
    width=None,
    opacity=None,
    background=False,
    family=True,
):
    if family:
        for submobject in self.submobjects:
            submobject.set_stroke(color, width, opacity, background, family)
    if background:
        array_name = "background_stroke_rgbs"
        width_name = "background_stroke_width"
        opacity_name = "background_stroke_opacity"
    else:
        array_name = "stroke_rgbs"
        width_name = "stroke_width"
        opacity_name = "stroke_opacity"
    self.update_rgbs_array(array_name, color, opacity)
    if width is not None:
        setattr(self, width_name, width)
    if opacity is not None:
        setattr(self, opacity_name, opacity)
    if color is not None and background:
        self.background_stroke_color = color
    return self

def set_background_stroke(self, **kwargs):
    kwargs["background"] = True
    self.set_stroke(**kwargs)
    return self

def set_style(
    self,
    fill_color=None,
    fill_opacity=None,
    stroke_color=None,
    stroke_width=None,
    stroke_opacity=None,
    background_stroke_color=None,
    background_stroke_width=None,
    background_stroke_opacity=None,
    sheen_factor=None,
    sheen_direction=None,
    background_image=None,
    family=True,
):
    self.set_fill(color=fill_color, opacity=fill_opacity, family=family)
    self.set_stroke(
        color=stroke_color,
        width=stroke_width,
        opacity=stroke_opacity,
        family=family,
    )
    self.set_background_stroke(
        color=background_stroke_color,
        width=background_stroke_width,
        opacity=background_stroke_opacity,
        family=family,
    )
    if sheen_factor:
        self.set_sheen(
            factor=sheen_factor,
            direction=sheen_direction,
            family=family,
        )
    if background_image:
        self.color_using_background_image(background_image)
    return self

def get_style(self, simple=False):
    ret = {
        "stroke_opacity": self.get_stroke_opacity(),
        "stroke_width": self.get_stroke_width(),
    }

    if simple:
        ret["fill_color"] = colour.rgb2hex(self.get_fill_color().get_rgb())
        ret["fill_opacity"] = self.get_fill_opacity()
        ret["stroke_color"] = colour.rgb2hex(self.get_stroke_color().get_rgb())
    else:
        ret["fill_color"] = self.get_fill_colors()
        ret["fill_opacity"] = self.get_fill_opacities()
        ret["stroke_color"] = self.get_stroke_colors()
        ret["background_stroke_color"] = self.get_stroke_colors(background=True)
        ret["background_stroke_width"] = self.get_stroke_width(background=True)
        ret["background_stroke_opacity"] = self.get_stroke_opacity(background=True)
        ret["sheen_factor"] = self.get_sheen_factor()
        ret["sheen_direction"] = self.get_sheen_direction()
        ret["background_image"] = self.get_background_image()

    return ret

def match_style(self, vmobject, family=True):
    self.set_style(**vmobject.get_style(), family=False)

```

```

if family:
    # Does its best to match up subobject lists, and
    # match styles accordingly
    submobs1, submobs2 = self.submobjects, vmobject.submobjects
    if len(submobs1) == 0:
        return self
    elif len(submobs2) == 0:
        submobs2 = [vmobject]
    for sm1, sm2 in zip(*make_even(submobs1, submobs2)):
        sm1.match_style(sm2)
    return self

def set_color(self, color, family=True):
    self.set_fill(color, family=family)
    self.set_stroke(color, family=family)
    return self

def set_opacity(self, opacity, family=True):
    self.set_fill(opacity=opacity, family=family)
    self.set_stroke(opacity=opacity, family=family)
    self.set_stroke(opacity=opacity, family=family, background=True)
    return self

def fade(self, darkness=0.5, family=True):
    factor = 1.0 - darkness
    self.set_fill(opacity=factor * self.get_fill_opacity(), family=False)
    self.set_stroke(opacity=factor * self.get_stroke_opacity(), family=False)
    self.set_background_stroke(
        opacity=factor * self.get_stroke_opacity(background=True),
        family=False,
    )
    super().fade(darkness, family)
    return self

def get_fill_rgbas(self):
    try:
        return self.fill_rgbas
    except AttributeError:
        return np.zeros((1, 4))

def get_fill_color(self):
    """
    If there are multiple colors (for gradient)
    this returns the first one
    """
    return self.get_fill_colors()[0]

fill_color = property(get_fill_color, set_fill)

def get_fill_opacity(self):
    """
    If there are multiple opacities, this returns the
    first
    """
    return self.get_fill_opacities()[0]

def get_fill_colors(self):
    return [
        colour.Color(rgb=rgba[:3]) if rgba.any() else None
        for rgba in self.get_fill_rgbas()
    ]

def get_fill_opacities(self):
    return self.get_fill_rgbas()[:, 3]

def get_stroke_rgbas(self, background=False):
    try:
        if background:
            rgbas = self.background_stroke_rgbas
        else:
            rgbas = self.stroke_rgbas
        return rgbas
    except AttributeError:
        return np.zeros((1, 4))

def get_stroke_color(self, background=False):
    return self.get_stroke_colors(background)[0]

stroke_color = property(get_stroke_color, set_stroke)

def get_stroke_width(self, background=False):
    if background:
        width = self.background_stroke_width
    else:
        width = self.stroke_width
        if isinstance(width, str):
            width = int(width)
    return max(0, width)

def get_stroke_opacity(self, background=False):
    return self.get_stroke_opacities(background)[0]

def get_stroke_colors(self, background=False):
    return [
        colour.Color(rgb=rgba[:3]) if rgba.any() else None
        for rgba in self.get_stroke_rgbas(background)
    ]

def get_stroke_opacities(self, background=False):
    return self.get_stroke_rgbas(background)[:, 3]

def get_color(self):
    if np.all(self.get_fill_opacities() == 0):
        return self.get_stroke_color()
    return self.get_fill_color()

color = property(get_color, set_color)

def set_sheen_direction(self, direction: np.ndarray, family=True):
    """Sets the direction of the applied sheen.

    Parameters
    -----
    direction
        Direction from where the gradient is applied.

    Examples
    -----
    Normal usage::

        Circle().set_sheen_direction(UP)

    See Also
    -----

```

```

-----
:meth:`~.VMobject.set_sheen`
:meth:`~.VMobject.rotate_sheen_direction`
"""

direction = np.array(direction)
if family:
    for submob in self.get_family():
        submob.sheen_direction = direction
else:
    self.sheen_direction = direction
return self

def rotate_sheen_direction(self, angle: float, axis: np.ndarray = OUT, family=True):
    """Rotates the direction of the applied sheen.

Parameters
-----
angle
    Angle by which the direction of sheen is rotated.
axis
    Axis of rotation.

Examples
-----
Normal usage::

    Circle().set_sheen_direction(UP).rotate_sheen_direction(PI)

See Also
-----
:meth:`~.VMobject.set_sheen_direction`
"""

if family:
    for submob in self.get_family():
        submob.sheen_direction = rotate_vector(
            submob.sheen_direction,
            angle,
            axis,
        )
else:
    self.sheen_direction = rotate_vector(self.sheen_direction, angle, axis)
return self

def set_sheen(self, factor: float, direction: np.ndarray = None, family=True):
    """Applies a color gradient from a direction.

Parameters
-----
factor
    The extent of lustre/gradient to apply. If negative, the gradient
    starts from black, if positive the gradient starts from white and
    changes to the current color.
direction
    Direction from where the gradient is applied.

Examples
-----
.. manim:: SetSheen
    :save_last_frame:

    class SetSheen(Scene):
        def construct(self):
            circle = Circle(fill_opacity=1).set_sheen(-0.3, DR)
            self.add(circle)
    """

if family:
    for submob in self.subobjects:
        submob.set_sheen(factor, direction, family)
self.sheen_factor = factor
if direction is not None:
    # family set to False because recursion will
    # already be handled above
    self.set_sheen_direction(direction, family=False)
# Reset color to put sheen_factor into effect
if factor != 0:
    self.set_stroke(self.get_stroke_color(), family=family)
    self.set_fill(self.get_fill_color(), family=family)
return self

def get_sheen_direction(self):
    return np.array(self.sheen_direction)

def get_sheen_factor(self):
    return self.sheen_factor

def get_gradient_start_and_end_points(self):
    if self.shade_in_3d:
        return get_3d_vmob_gradient_start_and_end_points(self)
    else:
        direction = self.get_sheen_direction()
        c = self.get_center()
        bases = np.array([
            [self.get_edge_center(vect) - c for vect in [RIGHT, UP, OUT]],
        ].transpose())
        offset = np.dot(bases, direction)
        return (c - offset, c + offset)

def color_using_background_image(self, background_image: Image | str):
    self.background_image = background_image
    self.set_color(WHITE)
    for submob in self.subobjects:
        submob.color_using_background_image(background_image)
    return self

def get_background_image(self) -> Image | str:
    return self.background_image

def match_background_image(self, vmobject):
    self.color_using_background_image(vmobject.get_background_image())
    return self

def set_shade_in_3d(self, value=True, z_index_as_group=False):
    for submob in self.get_family():
        submob.shade_in_3d = value
        if z_index_as_group:
            submob.z_index_group = self
    return self

def set_points(self, points):
    self.points = np.array(points)
    return self

```

```

def resize_points(
    self,
    new_length: int,
    resize_func: Callable[[np.ndarray, int], np.ndarray] = resize_array,
):
    """Resize the array of anchor points and handles to have
    the specified size.

    Parameters
    -----
    new_length
        The new (total) number of points.
    resize_func
        A function mapping a Numpy array (the points) and an integer
        (the target size) to a Numpy array. The default implementation
        is based on Numpy's ``resize`` function.

    """
    if new_length != len(self.points):
        self.points = resize_func(self.points, new_length)
    return self

def set_anchors_and_handles(
    self,
    anchors1: Sequence[float],
    handles1: Sequence[float],
    handles2: Sequence[float],
    anchors2: Sequence[float],
):
    """Given two sets of anchors and handles, process them to set them as anchors
    and handles of the VMobject.

    anchors1[i], handles1[i], handles2[i] and anchors2[i] define the i-th bezier
    curve of the vmobject. There are four hardcoded parameters and this is a
    problem as it makes the number of points per cubic curve unchangeable from 4
    (two anchors and two handles).

    Returns
    -----
    :class:`VMobject` ``self``

    """
    assert len(anchors1) == len(handles1) == len(handles2) == len(anchors2)
    nppcc = self.n_points_per_cubic_curve # 4
    total_len = nppcc * len(anchors1)
    self.points = np.zeros((total_len, self.dim))
    # the following will, from the four sets, dispatch them in points such that
    # self.points = [
    #     anchors1[0], handles1[0], handles2[0], anchors1[0], anchors1[1],
    #     handles1[1], ...
    # ]
    arrays = [anchors1, handles1, handles2, anchors2]
    for index, array in enumerate(arrays):
        self.points[index::nppcc] = array
    return self

def clear_points(self):
    self.points = np.zeros((0, self.dim))

def append_points(self, new_points):
    # TODO, check that number new points is a multiple of 4?
    # or else that if len(self.points) % 4 == 1, then
    # len(new_points) % 4 == 3?
    self.points = np.append(self.points, new_points, axis=0)
    return self

def start_new_path(self, point):
    if len(self.points) % 4 != 0:
        # close the open path by appending the last
        # start anchor sufficiently often
        last_anchor = self.get_start_anchors()[-1]
        for _ in range(4 - (len(self.points) % 4)):
            self.append_points([last_anchor])
    self.append_points([point])
    return self

def add_cubic_bezier_curve(
    self,
    anchor1: np.ndarray,
    handle1: np.ndarray,
    handle2: np.ndarray,
    anchor2,
) -> None:
    # TODO, check the len(self.points) % 4 == 0?
    self.append_points([anchor1, handle1, handle2, anchor2])

def add_cubic_bezier_curves(self, curves):
    self.append_points(curves.flatten())

def add_cubic_bezier_curve_to(
    self,
    handle1: np.ndarray,
    handle2: np.ndarray,
    anchor: np.ndarray,
):
    """Add cubic bezier curve to the path.

    NOTE : the first anchor is not a parameter as by default the end of the last sub-path!

    Parameters
    -----
    handle1
        first handle
    handle2
        second handle
    anchor
        anchor

    Returns
    -----
    :class:`VMobject` ``self``

    """
    self.throw_error_if_no_points()
    new_points = [handle1, handle2, anchor]
    if self.has_new_path_started():
        self.append_points(new_points)
    else:
        self.append_points([self.get_last_point()] + new_points)
    return self

def add_quadratic_bezier_curve_to(
    self,
)

```

```

handle: np.ndarray,
anchor: np.ndarray,
):
    """Add Quadratic bezier curve to the path.

    Returns
    ------
    :class:`VMobject`
        ``self``

    """
    # How does one approximate a quadratic with a cubic?
    # refer to the Wikipedia page on Bezier curves
    # https://en.wikipedia.org/wiki/Bezier_curve#Degree_elevation, accessed Jan 20, 2021
    # 1. Copy the end points, and then
    # 2. Place the 2 middle control points 2/3 along the line segments
    # from the end points to the quadratic curve's middle control point.
    # I think that's beautiful.
    self.add_cubic_bezier_curve_to(
        2 / 3 * handle + 1 / 3 * self.get_last_point(),
        2 / 3 * handle + 1 / 3 * anchor,
        anchor,
    )
    return self

def add_line_to(self, point: np.ndarray):
    """Add a straight line from the last point of VMobject to the given point.

    Parameters
    ------
    point
        end of the straight line.

    Returns
    ------
    :class:`VMobject`
        ``self``

    """
    nppcc = self.n_points_per_cubic_curve
    self.add_cubic_bezier_curve_to(
        *(
            interpolate(self.get_last_point(), point, a)
            for a in np.linspace(0, 1, nppcc)[1:]
        )
    )
    return self

def add_smooth_curve_to(self, *points: np.array):
    """Creates a smooth curve from given points and add it to the VMobject. If two points are passed in, the first is interpreted as a handle, the second as an anchor.

    Parameters
    ------
    points
        Points (anchor and handle, or just anchor) to add a smooth curve from

    Returns
    ------
    :class:`VMobject`
        ``self``

    Raises
    ------
    ValueError
        If 0 or more than 2 points are given.

    """
    # TODO remove the value error and just add two parameters with one optional
    if len(points) == 1:
        handle2 = None
        new_anchor = points[0]
    elif len(points) == 2:
        handle2, new_anchor = points
    else:
        name = sys._getframe(0).f_code.co_name
        raise ValueError(f"Only call {name} with 1 or 2 points")

    if self.has_new_path_started():
        self.add_line_to(new_anchor)
    else:
        self.throw_error_if_no_points()
        last_h2, last_a2 = self.points[-2:]
        last_tangent = last_a2 - last_h2
        handle1 = last_a2 + last_tangent
        if handle2 is None:
            to_anchor_vect = new_anchor - last_a2
            new_tangent = rotate_vector(last_tangent, PI, axis=to_anchor_vect)
            handle2 = new_anchor - new_tangent
        self.append_points([last_a2, handle1, handle2, new_anchor])
    return self

def has_new_path_started(self):
    nppcc = self.n_points_per_cubic_curve # 4
    # A new path starting is defined by a control point which is not part of a bezier subcurve.
    return len(self.points) % nppcc == 1

def get_last_point(self):
    return self.points[-1]

def is_closed(self):
    # TODO use consider_points_equals_2d ?
    return self.consider_points_equals(self.points[0], self.points[-1])

def close_path(self):
    if not self.is_closed():
        self.add_line_to(self.get_subpaths()[-1][0])

def add_points_as_corners(self, points: np.ndarray) -> VMobject:
    for point in points:
        self.add_line_to(point)
    return points

def set_points_as_corners(self, points: Sequence[float]):
    """Given an array of points, set them as corner of the vmobject.

    To achieve that, this algorithm sets handles aligned with the anchors such that the resultant bezier curve will be the segment between the two anchors.

    Parameters
    ------
    points
        Array of points that will be set as corners.

    Returns
    ------

```

```

-----
:klass: `VMobject`
    ``self``
"""
nppcc = self.n_points_per_cubic_curve
points = np.array(points)
# This will set the handles aligned with the anchors.
# Id est, a bezier curve will be the segment from the two anchors such that the handles belongs to this segment.
self.set_anchors_and_handles(
    *(interpolate(points[:-1], points[1:], a) for a in np.linspace(0, 1, nppcc)))
)
return self

def set_points_smoothly(self, points):
    self.set_points_as_corners(points)
    self.make_smooth()
    return self

def change_anchor_mode(self, mode: str):
    """Changes the anchor mode of the bezier curves. This will modify the handles.

    There can be only two modes, "jagged", and "smooth".
    """
    Returns
    -----
:klass: `VMobject`
    ``self``
"""
assert mode in ["jagged", "smooth"]
nppcc = self.n_points_per_cubic_curve
for submob in self.family_members_with_points():
    subpaths = submob.get_subpaths()
    submob.clear_points()
    # A subpath can be composed of several bezier curves.
    for subpath in subpaths:
        # This will retrieve the anchors of the subpath, by selecting every n element in the array subpath
        # The append is needed as the last element is not reached when slicing with numpy.
        anchors = np.append(subpath[:nppcc], subpath[-1:], 0)
        if mode == "smooth":
            h1, h2 = get_smooth_handle_points(anchors)
        elif mode == "jagged":
            # The following will make the handles aligned with the anchors, thus making the bezier curve a segment
            a1 = anchors[:-1]
            a2 = anchors[1:]
            h1 = interpolate(a1, a2, 1.0 / 3)
            h2 = interpolate(a1, a2, 2.0 / 3)
        new_subpath = np.array(subpath)
        new_subpath[1::nppcc] = h1
        new_subpath[2::nppcc] = h2
        submob.append_points(new_subpath)
    return self

def make_smooth(self):
    return self.change_anchor_mode("smooth")

def make_jagged(self):
    return self.change_anchor_mode("jagged")

def add_subpath(self, points: np.ndarray):
    assert len(points) % 4 == 0
    self.points = np.append(self.points, points, axis=0)
    return self

def append_vectorized_mobject(self, vectorized_mobject):
    new_points = list(vectorized_mobject.points)

    if self.has_new_path_started():
        # Remove last point, which is starting
        # a new path
        self.points = self.points[:-1]
        self.append_points(new_points)

def apply_function(self, function):
    factor = self.pre_function_handle_to_anchor_scale_factor
    self.scale_handle_to_anchor_distances(factor)
    super().apply_function(function)
    self.scale_handle_to_anchor_distances(1.0 / factor)
    if self.make_smooth_after_applying_functions:
        self.make_smooth()
    return self

def rotate(
    self,
    angle: float,
    axis: np.ndarray = OUT,
    about_point: Sequence[float] | None = None,
    **kwargs,
):
    self.rotate_sheen_direction(angle, axis)
    super().rotate(angle, axis, about_point, **kwargs)
    return self

def scale_handle_to_anchor_distances(self, factor: float):
    """If the distance between a given handle point H and its associated
    anchor point A is d, then it changes H to be a distances factor*d
    away from A, but so that the line from A to H doesn't change.
    This is mostly useful in the context of applying a (differentiable)
    function, to preserve tangency properties. One would pull all the
    handles closer to their anchors, apply the function then push them out
    again.
    """
    Parameters
    -----
    factor
        The factor used for scaling.
    Returns
    -----
:klass: `VMobject`
    ``self``
"""
for submob in self.family_members_with_points():
    if len(submob.points) < self.n_points_per_cubic_curve:
        # The case that a bezier quad is not complete (there is no bezier curve as there is not enough control points.)
        continue
    a1, h1, h2, a2 = submob.get_anchors_and_handles()
    a1_to_h1 = h1 - a1
    a2_to_h2 = h2 - a2
    new_h1 = a1 + factor * a1_to_h1
    new_h2 = a2 + factor * a2_to_h2
    submob.set_anchors_and_handles(a1, new_h1, new_h2, a2)
return self

```

```

#
def consider_points_equals(self, p0, p1):
    return np.allclose(p0, p1, atol=self.tolerance_for_point_equality)

def consider_points_equals_2d(self, p0: np.ndarray, p1: np.ndarray) -> bool:
    """Determine if two points are close enough to be considered equal.

    This uses the algorithm from np.isclose(), but expanded here for the
    2D point case. NumPy is overkill for such a small question.

    Parameters
    -----
    p0
        first point
    p1
        second point

    Returns
    -----
    bool
        whether two points considered close.
    """
    rtol = 1.0e-5 # default from np.isclose()
    atol = self.tolerance_for_point_equality
    if abs(p0[0] - p1[0]) > atol + rtol * abs(p1[0]):
        return False
    if abs(p0[1] - p1[1]) > atol + rtol * abs(p1[1]):
        return False
    return True

# Information about line
def get_cubic_bezier_tuples_from_points(self, points):
    return np.array(list(self.gen_cubic_bezier_tuples_from_points(points)))

def gen_cubic_bezier_tuples_from_points(self, points: np.ndarray) -> tuple:
    """Returns the bezier tuples from an array of points.

    self.points is a list of the anchors and handles of the bezier curves of the mobject (ie [anchor1, handle1, handle2, anchor2, anchor3 ...])
    This algorithm basically retrieve them by taking an element every n, where n is the number of control points
    of the bezier curve.

    Parameters
    -----
    points
        Points from which control points will be extracted.

    Returns
    -----
    typing.Tuple
        Bezier control points.
    """
    nppcc = self.n_points_per_cubic_curve
    remainder = len(points) % nppcc
    points = points[: len(points) - remainder]
    # Basically take every nppcc element.
    return (points[i : i + nppcc] for i in range(0, len(points), nppcc))

def get_cubic_bezier_tuples(self):
    return self.get_cubic_bezier_tuples_from_points(self.points)

def _gen_subpaths_from_points(
    self,
    points: np.ndarray,
    filter_func: typing.Callable[[int], bool],
) -> tuple:
    """Given an array of points defining the bezier curves of the vmobject, return subpaths formed by these points.
    Here, Two bezier curves form a path if at least two of their anchors are evaluated True by the relation defined by filter_func.

    The algorithm every bezier tuple (anchors and handles) in ``self.points`` (by regrouping each n elements, where
    n is the number of points per cubic curve)), and evaluate the relation between two anchors with filter_func.
    NOTE : The filter_func takes an int n as parameter, and will evaluate the relation between points[n] and points[n - 1]. This should probably be changed so
    the function takes two points as parameters.

    Parameters
    -----
    points
        points defining the bezier curve.
    filter_func
        Filter-func defining the relation.

    Returns
    -----
    typing.Tuple
        subpaths formed by the points.
    """
    nppcc = self.n_points_per_cubic_curve
    filtered = filter(filter_func, range(nppcc, len(points), nppcc))
    split_indices = [0] + list(filtered) + [len(points)]
    return (
        points[i1:i2]
        for i1, i2 in zip(split_indices, split_indices[1:])
        if (i2 - i1) >= nppcc
    )

def get_subpaths_from_points(self, points):
    return list(
        self._gen_subpaths_from_points(
            points,
            lambda n: not self.consider_points_equals(points[n - 1], points[n]),
        ),
    )

def gen_subpaths_from_points_2d(self, points):
    return self._gen_subpaths_from_points(
        points,
        lambda n: not self.consider_points_equals_2d(points[n - 1], points[n]),
    )

def get_subpaths(self) -> tuple:
    """Returns subpaths formed by the curves of the VMobject.

    Subpaths are ranges of curves with each pair of consecutive curves having their end/start points coincident.

    Returns
    -----
    typing.Tuple
        subpaths.
    """
    return self.get_subpaths_from_points(self.points)

def get_nth_curve_points(self, n: int) -> np.ndarray:
    """Returns the points defining the nth curve of the vmobject.

```

```

Parameters
-----
n
    index of the desired bezier curve.

Returns
-----
np.ndarray
    points definininf the nth bezier curve (anchors, handles)
"""
assert n < self.get_num_curves()
nppcc = self.n_points_per_cubic_curve
return self.points[nppcc * n : nppcc * (n + 1)]

def get_nth_curve_function(self, n: int) -> typing.Callable[[float], np.ndarray]:
    """Returns the expression of the nth curve.

Parameters
-----
n
    index of the desired curve.

Returns
-----
typing.Callable[float]
    expression of the nth bezier curve.
"""
    return bezier(self.get_nth_curve_points(n))

def get_nth_curve_length_pieces(
    self,
    n: int,
    sample_points: int | None = None,
) -> np.ndarray:
    """Returns the array of short line lengths used for length approximation.

Parameters
-----
n
    The index of the desired curve.
sample_points
    The number of points to sample to find the length.

Returns
-----
np.ndarray
    The short length-pieces of the nth curve.
"""
    if sample_points is None:
        sample_points = 10

    curve = self.get_nth_curve_function(n)
    points = np.array([curve(a) for a in np.linspace(0, 1, sample_points)])
    diffs = points[1:] - points[:-1]
    norms = np.linalg.norm(diffs, axis=1)

    return norms

def get_nth_curve_length(
    self,
    n: int,
    sample_points: int | None = None,
) -> float:
    """Returns the (approximate) length of the nth curve.

Parameters
-----
n
    The index of the desired curve.
sample_points
    The number of points to sample to find the length.

Returns
-----
length : :class:`float`
    The length of the nth curve.
"""
    _, length = self.get_nth_curve_function_with_length(n, sample_points)

    return length

def get_nth_curve_function_with_length(
    self,
    n: int,
    sample_points: int | None = None,
) -> tuple[typing.Callable[[float], np.ndarray], float]:
    """Returns the expression of the nth curve along with its (approximate) length.

Parameters
-----
n
    The index of the desired curve.
sample_points
    The number of points to sample to find the length.

Returns
-----
curve : typing.Callable[[float], np.ndarray]
    The function for the nth curve.
length : :class:`float`
    The length of the nth curve.
"""
    curve = self.get_nth_curve_function(n)
    norms = self.get_nth_curve_length_pieces(n, sample_points=sample_points)
    length = np.sum(norms)

    return curve, length

def get_num_curves(self) -> int:
    """Returns the number of curves of the vmobject.

Returns
-----
int
    number of curves. of the vmobject.
"""
    nppcc = self.n_points_per_cubic_curve
    return len(self.points) // nppcc

def get_curve_functions(
    self,

```

```

) -> typing.Iterable[typing.Callable[[float], np.ndarray]]:
    """Gets the functions for the curves of the mobject.

    Returns
    ------
    typing.Iterable[typing.Callable[[float], np.ndarray]]
        The functions for the curves.
    """

    num_curves = self.get_num_curves()

    for n in range(num_curves):
        yield self.get_nth_curve_function(n)

def get_curve_functions_with_lengths(
    self, **kwargs
) -> typing.Iterable[tuple[typing.Callable[[float], np.ndarray], float]]:
    """Gets the functions and lengths of the curves for the mobject.

    Parameters
    -----
    **kwargs
        The keyword arguments passed to :meth:`get_nth_curve_function_with_length`


    Returns
    ------
    typing.Iterable[tuple[typing.Callable[[float], np.ndarray], float]]
        The functions and lengths of the curves.
    """

    num_curves = self.get_num_curves()

    for n in range(num_curves):
        yield self.get_nth_curve_function_with_length(n, **kwargs)

def point_from_proportion(self, alpha: float) -> np.ndarray:
    """Gets the point at a proportion along the path of the :class:`VMobject`.

    Parameters
    -----
    alpha
        The proportion along the the path of the :class:`VMobject`.

    Returns
    ------
    :class:`numpy.ndarray`
        The point on the :class:`VMobject`.

    Raises
    -----
    :exc:`ValueError`
        If ``alpha`` is not between 0 and 1.
    :exc:`Exception`
        If the :class:`VMobject` has no points.
    """

    if alpha < 0 or alpha > 1:
        raise ValueError(f"Alpha {alpha} not between 0 and 1.")

    self.throw_error_if_no_points()
    if alpha == 1:
        return self.points[-1]

    curves_and_lengths = tuple(self.get_curve_functions_with_lengths())

    target_length = alpha * sum(length for _, length in curves_and_lengths)
    current_length = 0

    for curve, length in curves_and_lengths:
        if current_length + length >= target_length:
            if length != 0:
                residue = (target_length - current_length) / length
            else:
                residue = 0

            return curve(residue)

        current_length += length

def proportion_from_point(
    self,
    point: typing.Iterable[float | int],
) -> float:
    """Returns the proportion along the path of the :class:`VMobject` a particular given point is at.

    Parameters
    -----
    point
        The Cartesian coordinates of the point which may or may not lie on the :class:`VMobject`.

    Returns
    ------
    float
        The proportion along the path of the :class:`VMobject`.

    Raises
    -----
    :exc:`ValueError`
        If ``point`` does not lie on the curve.
    :exc:`Exception`
        If the :class:`VMobject` has no points.
    """

    self.throw_error_if_no_points()

    # Iterate over each bezier curve that the ``VMobject`` is composed of, checking
    # if the point lies on that curve. If it does not lie on that curve, add
    # the whole length of the curve to ``target_length`` and move onto the next
    # curve. If the point does lie on the curve, add how far along the curve
    # the point is to ``target_length``.
    # Then, divide ``target_length`` by the total arc length of the shape to get
    # the proportion along the ``VMobject`` the point is at.

    num_curves = self.get_num_curves()
    total_length = self.get_arc_length()
    target_length = 0
    for n in range(num_curves):
        control_points = self.get_nth_curve_points(n)
        length = self.get_nth_curve_length(n)
        proportions_along_bezier = proportions_along_bezier_curve_for_point(
            point,
            control_points,
        )

```

```

if len(proportions_along_bezier) > 0:
    proportion_along_nth_curve = max(proportions_along_bezier)
    target_length += length * proportion_along_nth_curve
    break
target_length += length
else:
    raise ValueError(f"Point {point} does not lie on this curve.")

alpha = target_length / total_length

return alpha

def get_anchors_and_handles(self) -> typing.Iterable[np.ndarray]:
    """Returns anchors1, handles1, handles2, anchors2,
    where (anchors1[i], handles1[i], handles2[i], anchors2[i])
    will be four points defining a cubic bezier curve
    for any i in range(0, len(anchors1))

    Returns
    ------
    typing.Iterable[np.ndarray]
        Iterable of the anchors and handles.
    """
    nppcc = self.n_points_per_cubic_curve
    return [self.points[i::nppcc] for i in range(nppcc)]

def get_start_anchors(self) -> np.ndarray:
    """Returns the start anchors of the bezier curves.

    Returns
    ------
    np.ndarray
        Starting anchors
    """
    return self.points[0 :: self.n_points_per_cubic_curve]

def get_end_anchors(self) -> np.ndarray:
    """Return the end anchors of the bezier curves.

    Returns
    ------
    np.ndarray
        Starting anchors
    """
    nppcc = self.n_points_per_cubic_curve
    return self.points[nppcc - 1 :: nppcc]

def get_anchors(self) -> np.ndarray:
    """Returns the anchors of the curves forming the VMobject.

    Returns
    ------
    np.ndarray
        The anchors.
    """
    if self.points.shape[0] == 1:
        return self.points
    return np.array(
        list(it.chain(*zip(self.get_start_anchors(), self.get_end_anchors()))),
    )

def get_points_defining_boundary(self):
    # Probably returns all anchors, but this is weird regarding the name of the method.
    return np.array(list(it.chain(*sm.get_anchors() for sm in self.get_family())))

def get_arc_length(self, sample_points_per_curve: int | None = None) -> float:
    """Return the approximated length of the whole curve.

    Parameters
    -----
    sample_points_per_curve
        Number of sample points per curve used to approximate the length. More points result in a better approximation.

    Returns
    ------
    float
        The length of the :class:`VMobject`.
    """

    return sum(
        length
        for _, length in self.get_curve_functions_with_lengths(
            sample_points=sample_points_per_curve,
        )
    )

# Alignment
def align_points(self, vmobject: VMobject):
    """Adds points to self and vmobject so that they both have the same number of subpaths, with
    corresponding subpaths each containing the same number of points.

    Points are added either by subdividing curves evenly along the subpath, or by creating new subpaths consisting
    of a single point repeated.

    Parameters
    -----
    vmobject
        The object to align points with.

    Returns
    -----
    :class:`VMobject`
        ``self``

    """
    self.align_rgbs(vmobject)
    # TODO: This shortcut can be a bit over eager. What if they have the same length, but different subpath lengths?
    if self.get_num_points() == vmobject.get_num_points():
        return

    for mob in self, vmobject:
        # If there are no points, add one to
        # wherever the "center" is
        if mob.has_no_points():
            mob.start_new_path(mob.get_center())
        # If there's only one point, turn it into
        # a null curve
        if mob.has_new_path_started():
            mob.add_line_to(mob.get_last_point())

    # Figure out what the subpaths are
    subpaths1 = self.get_subpaths()
    subpaths2 = vmobject.get_subpaths()
    n_subpaths = max(len(subpaths1), len(subpaths2))

```

```

# Start building new ones
new_path1 = np.zeros((0, self.dim))
new_path2 = np.zeros((0, self.dim))

nppcc = self.n_points_per_cubic_curve

def get_nth_subpath(path_list, n):
    if n >= len(path_list):
        # Create a null path at the very end
        return [path_list[-1]] * nppcc
    path = path_list[n]
    # Check for useless points at the end of the path and remove them
    # https://github.com/ManimCommunity/manim/issues/1959
    while len(path) > nppcc:
        # If the last nppcc points are all equal to the preceding point
        if self.consider_points_equals(path[-nppcc:], path[-nppcc - 1]):
            path = path[:-nppcc]
        else:
            break
    return path

for n in range(n_subpaths):
    # For each pair of subpaths, add points until they are the same length
    sp1 = get_nth_subpath(subpaths1, n)
    sp2 = get_nth_subpath(subpaths2, n)
    diff1 = max(0, (len(sp2) - len(sp1)) // nppcc)
    diff2 = max(0, (len(sp1) - len(sp2)) // nppcc)
    sp1 = self.insert_n_curves_to_point_list(diff1, sp1)
    sp2 = self.insert_n_curves_to_point_list(diff2, sp2)
    new_path1 = np.append(new_path1, sp1, axis=0)
    new_path2 = np.append(new_path2, sp2, axis=0)
self.set_points(new_path1)
vmobject.set_points(new_path2)
return self

def insert_n_curves(self, n: int):
    """Inserts n curves to the bezier curves of the vmobject.

    Parameters
    -----
    n
        Number of curves to insert.

    Returns
    -----
    :class:`VMobject`
        ``self``

    """
    new_path_point = None
    if self.has_new_path_started():
        new_path_point = self.get_last_point()

    new_points = self.insert_n_curves_to_point_list(n, self.points)
    self.set_points(new_points)

    if new_path_point is not None:
        self.append_points([new_path_point])
    return self

def insert_n_curves_to_point_list(self, n: int, points: np.ndarray) -> np.ndarray:
    """Given an array of k points defining a bezier curves (anchors and handles), returns points defining exactly k + n bezier curves.

    Parameters
    -----
    n
        Number of desired curves.
    points
        Starting points.

    Returns
    -----
    np.ndarray
        Points generated.

    """
    if len(points) == 1:
        nppcc = self.n_points_per_cubic_curve
        return np.repeat(points, nppcc * n, 0)
    bezier_quads = self.get_cubic_bezier_tuples_from_points(points)
    curr_num = len(bezier_quads)
    target_num = curr_num + n
    # This is an array with values ranging from 0
    # up to curr_num, with repeats such that
    # it's total length is target_num. For example,
    # with curr_num = 10, target_num = 15, this would
    # be [0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9]
    repeat_indices = (np.arange(target_num, dtype="i") * curr_num) // target_num

    # If the nth term of this list is k, it means
    # that the nth curve of our path should be split
    # into k pieces.
    # In the above example our array had the following elements
    # [0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9]
    # We have two 0s, one 1, two 2s and so on.
    # The split factors array would hence be:
    # [2, 1, 2, 1, 2, 1, 2, 1]
    split_factors = np.zeros(curr_num, dtype="i")
    for val in repeat_indices:
        split_factors[val] += 1

    new_points = np.zeros((0, self.dim))
    for quad, sf in zip(bezier_quads, split_factors):
        # What was once a single cubic curve defined
        # by "quad" will now be broken into sf
        # smaller cubic curves
        alphas = np.linspace(0, 1, sf + 1)
        for a1, a2 in zip(alphas, alphas[1:]):
            new_points = np.append(
                new_points,
                partial_bezier_points(quad, a1, a2),
                axis=0,
            )
    return new_points

def align_rgbs(self, vmobject):
    attrs = ["fill_rgbs", "stroke_rgbs", "background_stroke_rgbs"]
    for attr in attrs:
        a1 = getattr(self, attr)
        a2 = getattr(vmobject, attr)
        if len(a1) > len(a2):
            new_a2 = stretch_array_to_length(a2, len(a1))
            setattr(vmobject, attr, new_a2)
        elif len(a2) > len(a1):

```

```

new_a1 = stretch_array_to_length(a1, len(a2))
setattr(self, attr, new_a1)
return self

def get_point_mobject(self, center=None):
    if center is None:
        center = self.get_center()
    point = VectorizedPoint(center)
    point.match_style(self)
    return point

def interpolate_color(self, mobject1, mobject2, alpha):
    attrs = [
        "fill_rgbs",
        "stroke_rgbs",
        "background_stroke_rgbs",
        "stroke_width",
        "background_stroke_width",
        "sheen_direction",
        "sheen_factor",
    ]
    for attr in attrs:
        setattr(
            self,
            attr,
            interpolate(getattr(mobject1, attr), getattr(mobject2, attr), alpha),
        )
    if alpha == 1.0:
        setattr(self, attr, getattr(mobject2, attr))

def pointwise_become_partial(
    self,
    vmobject: VMobject,
    a: float,
    b: float,
):
    """
    Given two bounds a and b, transforms the points of the self vmobject into the points of the vmobject
    passed as parameter with respect to the bounds. Points here stand for control points of the bezier curves (anchors and handles)

    Parameters
    -----
    vmobject
        The vmobject that will serve as a model.
    a
        upper-bound.
    b
        lower-bound

    Returns
    -----
    :class:`VMobject`
        ``self``
    """
    assert isinstance(vmobject, VMobject)
    # Partial curve includes three portions:
    # - A middle section, which matches the curve exactly
    # - A start, which is some ending portion of an inner cubic
    # - An end, which is the starting portion of a later inner cubic
    if a <= 0 and b >= 1:
        self.set_points(vmobject.points)
        return self
    bezier_quads = vmobject.get_cubic_bezier_tuples()
    num_cubics = len(bezier_quads)

    # The following two lines will compute which bezier curves of the given mobject need to be processed.
    # The residue basically indicates de proportion of the selected bezier curve that have to be selected.
    # Ex : if lower_index is 3, and lower_residue is 0.4, then the algorithm will append to the points 0.4 of the third bezier curve
    lower_index, lower_residue = integer_interpolate(0, num_cubics, a)
    upper_index, upper_residue = integer_interpolate(0, num_cubics, b)

    self.clear_points()
    if num_cubics == 0:
        return self
    if lower_index == upper_index:
        self.append_points(
            partial_bezier_points(
                bezier_quads[lower_index],
                lower_residue,
                upper_residue,
            ),
        )
    else:
        self.append_points(
            partial_bezier_points(bezier_quads[lower_index], lower_residue, 1),
        )
        for quad in bezier_quads[lower_index + 1 : upper_index]:
            self.append_points(quad)
        self.append_points(
            partial_bezier_points(bezier_quads[upper_index], 0, upper_residue),
        )
    return self

def get_subcurve(self, a: float, b: float) -> VMobject:
    """
    Returns the subcurve of the VMobject between the interval [a, b].
    The curve is a VMobject itself.

    Parameters
    -----
    a
        The lower bound.
    b
        The upper bound.

    Returns
    -----
    VMobject
        The subcurve between of [a, b]
    """
    if self.is_closed() and a > b:
        vmob = self.copy()
        vmob.pointwise_become_partial(self, a, 1)
        vmob2 = self.copy()
        vmob2.pointwise_become_partial(self, 0, b)
        vmob.append_vectorized_mobject(vmob2)
    else:
        vmob = self.copy()
        vmob.pointwise_become_partial(self, a, b)
    return vmob

def get_direction(self):
    """
    Uses :func:`~space_ops.shoelace_direction` to calculate the direction.
    The direction of points determines in which direction the

```

```

object is drawn, clockwise or counterclockwise.

Examples
-----
The default direction of a :class:`~.Circle` is counterclockwise::

    >>> from manim import Circle
    >>> Circle().get_direction()
    'CCW'

Returns
-----
:class:`str`
    Either ``"CW`` or ``"CCW``.

    return shoelace_direction(self.get_start_anchors())

def reverse_direction(self):
    """Reverts the point direction by inverting the point order.

Returns
-----
:class:`VMobject`
    Returns self.

Examples
-----
.. manim:: ChangeOfDirection

    class ChangeOfDirection(Scene):
        def construct(self):
            ccw = RegularPolygon(5)
            ccw.shift(LEFT)
            cw = RegularPolygon(5)
            cw.shift(RIGHT).reverse_direction()

            self.play(Create(ccw), Create(cw),
                      run_time=4)

        self.points = self.points[::-1]
        return self

def force_direction(self, target_direction: str):
    """Makes sure that points are either directed clockwise or
    counterclockwise.

Parameters
-----
target_direction
    Either ``"CW`` or ``"CCW``.

    if target_direction not in ("CW", "CCW"):
        raise ValueError("Invalid input for force_direction. Use "CW" or "CCW")
    if self.get_direction() != target_direction:
        # Since we already assured the input is CW or CCW,
        # and the directions don't match, we just reverse
        self.reverse_direction()
    return self

class VGroup(VMobject, metaclass=ConvertToOpenGL):
    """A group of vectorized mobjects.

This can be used to group multiple :class:`~.VMobject` instances together
in order to scale, move, ... them together.

Notes
-----
When adding the same mobject more than once, repetitions are ignored.
Use :meth:`.Mobject.copy` to create a separate copy which can then
be added to the group.

Examples
-----
To add :class:`~.VMobject`'s to a :class:`~.VGroup`, you can either use the
:meth:`~.VGroup.add` method, or use the `+` and `+=` operators. Similarly, you
can subtract elements of a VGroup via :meth:`~.VGroup.remove` method, or
`-` and `-=` operators:

    >>> from manim import Triangle, Square, VGroup
    >>> vg = VGroup()
    >>> triangle, square = Triangle(), Square()
    >>> vg.add(triangle)
    VGroup(Triangle)
    >>> vg + square  # a new VGroup is constructed
    VGroup(Triangle, Square)
    >>> vg          # not modified
    VGroup(Triangle)
    >>> vg += square; vg  # modifies vg
    VGroup(Triangle, Square)
    >>> vg.remove(triangle)
    VGroup(Square)
    >>> vg - square; # a new VGroup is constructed
    VGroup()
    >>> vg          # not modified
    VGroup(Square)
    >>> vg -= square; vg # modifies vg
    VGroup()

.. manim:: ArcShapeIris
    :save_last_frame:

    class ArcShapeIris(Scene):
        def construct(self):
            colors = [DARK_BROWN, BLUE_E, BLUE_D, BLUE_A, TEAL_B, GREEN_B, YELLOW_E]
            radius = [1 + rad * 0.1 for rad in range(len(colors))]

            circles_group = VGroup()

            # zip(radius, color) makes the iterator [(radius[i], color[i]) for i in range(radius)]
            circles_group.add(*[Circle(radius=rad, stroke_width=10, color=col)
                               for rad, col in zip(radius, colors)])
            self.add(circles_group)

        def __init__(self, *vmothers, **kwargs):
            super().__init__(**kwargs)
            self.add(*vmothers)

        def __repr__(self):
            return (

```

```

        self.__class__.__name__
        + "("
        + ", ".join(str(mob) for mob in self.submobjects)
        + ")"
    )

def __str__(self):
    return (
        f"{self.__class__.__name__} of {len(self.submobjects)} "
        f"submobject{'s' if len(self.submobjects) > 0 else ''}"
    )

def add(self, *vmobjects: VMobject):
    """Checks if all passed elements are an instance of VMobject and then add them to submobjects

    Parameters
    -----
    vmobjects
        List of VMobject to add

    Returns
    -----
    :class:`VGroup`

    Raises
    -----
    TypeError
        If one element of the list is not an instance of VMobject

    Examples
    -----
    .. manim:: AddToVGroup

        class AddToVGroup(Scene):
            def construct(self):
                circle_red = Circle(color=RED)
                circle_green = Circle(color=GREEN)
                circle_blue = Circle(color=BLUE)
                circle_red.shift(LEFT)
                circle_blue.shift(RIGHT)
                gr = VGroup(circle_red, circle_green)
                gr2 = VGroup(circle_blue) # Constructor uses add directly
                self.add(gr, gr2)
                self.wait()
                gr += gr2 # Add group to another
                self.play(
                    gr.animate.shift(DOWN),
                )
                gr -= gr2 # Remove group
                self.play( # Animate groups separately
                    gr.animate.shift(LEFT),
                    gr2.animate.shift(UP),
                )
                self.play( #Animate groups without modification
                    (gr+gr2).animate.shift(RIGHT)
                )
                self.play( # Animate group without component
                    (gr-circle_red).animate.shift(RIGHT)
                )
            """
            if not all(isinstance(m, (VMobject, OpenGLVMobject)) for m in vmobjects):
                raise TypeError("All submobjects must be of type VMobject")
            return super().add(*vmobjects)

    def __add__(self, vmobject):
        return VGroup(*self.submobjects, vmobject)

    def __iadd__(self, vmobject):
        return self.add(vmobject)

    def __sub__(self, vmobject):
        copy = VGroup(*self.submobjects)
        copy.remove(vmobject)
        return copy

    def __isub__(self, vmobject):
        return self.remove(vmobject)

    def __setitem__(self, key: int, value: VMobject | typing.Sequence[VMobject]):
        """Override the [] operator for item assignment.

        Parameters
        -----
        key
            The index of the submobject to be assigned
        value
            The vmobject value to assign to the key

        Returns
        -----
        None

        Tests
        -----
        Check that item assignment does not raise error:
        >>> vgroup = VGroup(VMobject())
        >>> new_obj = VMobject()
        >>> vgroup[0] = new_obj
        """
        if not all(isinstance(m, (VMobject, OpenGLVMobject)) for m in value):
            raise TypeError("All submobjects must be of type VMobject")
        self.submobjects[key] = value

    class VDict(VMobject, metaclass=ConvertToOpenGL):
        """A VGroup-like class, also offering submobject access by
        key, like a python dict

        Parameters
        -----
        mapping_or_iterable
            The parameter specifying the key-value mapping of keys and mobjects.
        show_keys
            Whether to also display the key associated with
            the mobject. This might be useful when debugging,
            especially when there are a lot of mobjects in the
            :class:`VDict`. Defaults to False.
        kwargs
            Other arguments to be passed to `Mobject`.

        Attributes
        -----
        show_keys : :class:`bool`
```

Whether to also display the key associated with the mobject. This might be useful when debugging, especially when there are a lot of mobjects in the :class:`'VDict'`. When displayed, the key is towards the left of the mobject.
 Defaults to False.

`submob_dict` : :class:`dict`
 Is the actual python dictionary that is used to bind the keys to the mobjects.

Examples

```
.. manim:: ShapesWithVDict

class ShapesWithVDict(Scene):
    def construct(self):
        square = Square().set_color(RED)
        circle = Circle().set_color(YELLOW).next_to(square, UP)

        # create dict from list of tuples each having key-mobject pair
        pairs = [("s", square), ("c", circle)]
        my_dict = VDict(pairs, show_keys=True)

        # display it just like a VGroup
        self.play(Create(my_dict))
        self.wait()

        text = Tex("Some text").set_color(GREEN).next_to(square, DOWN)

        # add a key-value pair by wrapping it in a single-element list of tuple
        # after attrs branch is merged, it will be easier like ` .add(t=text)`
        my_dict.add([("t", text)])
        self.wait()

        rect = Rectangle().next_to(text, DOWN)
        # can also do key assignment like a python dict
        my_dict["r"] = rect

        # access submobjects like a python dict
        my_dict["t"].set_color(PURPLE)
        self.play(my_dict["t"].animate.scale(3))
        self.wait()

        # also supports python dict styled reassignment
        my_dict["t"] = Tex("Some other text").set_color(BLUE)
        self.wait()

        # remove submobject by key
        my_dict.remove("t")
        self.wait()

        self.play(Uncreate(my_dict["s"]))
        self.wait()

        self.play(FadeOut(my_dict["c"]))
        self.wait()

        self.play(FadeOut(my_dict["r"], shift=DOWN))
        self.wait()

        # you can also make a VDict from an existing dict of mobjects
        plain_dict = {
            1: Integer(1).shift(DOWN),
            2: Integer(2).shift(2 * DOWN),
            3: Integer(3).shift(3 * DOWN),
        }

        vdict_from_plain_dict = VDict(plain_dict)
        vdict_from_plain_dict.shift(1.5 * (UP + LEFT))
        self.play(Create(vdict_from_plain_dict))

        # you can even use zip
        vdict_using_zip = VDict(zip(["s", "c", "r"], [Square(), Circle(), Rectangle()]))
        vdict_using_zip.shift(1.5 * RIGHT)
        self.play(Create(vdict_using_zip))
        self.wait()
    """

    def __init__(self,
                 mapping_or_iterable: (
                     typing.Mapping[typing.Hashable, VMobject]
                     | typing.Iterable[tuple[typing.Hashable, VMobject]]),
                 ) = {},
                 show_keys: bool = False,
                 **kwargs,
                 ):
        super().__init__(**kwargs)
        self.show_keys = show_keys
        self.submob_dict = {}
        self.add(mapping_or_iterable)

    def __repr__(self):
        return f'{self.__class__.__name__}({self.submob_dict})'

    def add(
            self,
            mapping_or_iterable: (
                typing.Mapping[typing.Hashable, VMobject]
                | typing.Iterable[tuple[typing.Hashable, VMobject]]),
            ),
            ):
        """Adds the key-value pairs to the :class:`'VDict'` object.

        Also, it internally adds the value to the `submobjects` :class:`list` of :class:`'~.Mobject'`, which is responsible for actual on-screen display.
        """
        Parameters
        -----
        mapping_or_iterable
            The parameter specifying the key-value mapping of keys and mobjects.

    Returns
    -----
    :class:`'VDict'`
        Returns the :class:`'VDict'` object on which this method was called.

    Examples
    -----
    Normal usage::

        square_obj = Square()
```

```

        my_dict.add([('s', square_obj)])
"""
for key, value in dict(mapping_or_iterable).items():
    self.add_key_value_pair(key, value)

return self

def remove(self, key: typing.Hashable):
    """Removes the mobject from the :class:`VDict` object having the key `key`

Also, it internally removes the mobject from the `submobjects` :class:`list` of :class:`~.Mobject`, (which is responsible for removing it from the screen)

Parameters
-----
key
    The key of the submoject to be removed.

Returns
-----
:class:`VDict`
    Returns the :class:`VDict` object on which this method was called.

Examples
-----
Normal usage::

    my_dict.remove('square')

"""
if key not in self.submob_dict:
    raise KeyError("The given key '%s' is not present in the VDict" % str(key))
super().remove(self.submob_dict[key])
del self.submob_dict[key]
return self

def __getitem__(self, key: typing.Hashable):
    """Override the [] operator for item retrieval.

Parameters
-----
key
    The key of the submoject to be accessed

Returns
-----
:class:`VMobject`
    The submobject corresponding to the key `key`

Examples
-----
Normal usage::

    self.play(Create(my_dict['s']))

submob = self.submob_dict[key]
return submob

def __setitem__(self, key: typing.Hashable, value: VMobject):
    """Override the [] operator for item assignment.

Parameters
-----
key
    The key of the submoject to be assigned
value
    The submobject to bind the key to

Returns
-----
None

Examples
-----
Normal usage::

    square_obj = Square()
    my_dict['sq'] = square_obj
"""
if key in self.submob_dict:
    self.remove(key)
self.add([(key, value)])

def __delitem__(self, key: typing.Hashable):
    """Override the del operator for deleting an item.

Parameters
-----
key
    The key of the submoject to be deleted

Returns
-----
None

Examples
-----
::

    >>> from manim import *
    >>> my_dict = VDict({'sq': Square()})
    >>> 'sq' in my_dict
    True
    >>> del my_dict['sq']
    >>> 'sq' in my_dict
    False

Notes
-----
Removing an item from a VDict does not remove that item from any Scene that the VDict is part of.

"""
del self.submob_dict[key]

def __contains__(self, key: typing.Hashable):
    """Override the in operator.

Parameters
-----
key
    The key to check membership of.

Returns
-----

```

```

-----
:class:`bool`

Examples
-----
::

    >>> from manim import *
    >>> my_dict = VDict({'sq': Square()})
    >>> 'sq' in my_dict
    True

"""
return key in self.submob_dict

def get_all_submobjects(self):
    """To get all the submobjects associated with a particular :class:`VDict` object

    Returns
    -----
    :class:`dict_values`
        All the submobjects associated with the :class:`VDict` object

    Examples
    -----
    Normal usage::

        for submob in my_dict.get_all_submobjects():
            self.play(Create(submob))

    submobjects = self.submob_dict.values()
    return submobjects

def add_key_value_pair(self, key: typing.Hashable, value: VMobject):
    """A utility function used by :meth:`add` to add the key-value pair to :attr:`submob_dict`. Not really meant to be used externally.

    Parameters
    -----
    key
        The key of the submobject to be added.
    value
        The mobobject associated with the key

    Returns
    -----
    None

    Raises
    -----
    TypeError
        If the value is not an instance of VMobject

    Examples
    -----
    Normal usage::

        square_obj = Square()
        self.add_key_value_pair('s', square_obj)

    """
    if not isinstance(value, (VMobject, OpenGLVMobject)):
        raise TypeError("All submobjects must be of type VMobject")
    mob = value
    if self.show_keys:
        # This import is here and not at the top to avoid circular import
        from manim.mobject.text.tex_mobject import Tex
        key_text = Tex(str(key)).next_to(value, LEFT)
        mob.add(key_text)
    self.submob_dict[key] = mob
    super().add(value)

class VectorizedPoint(VMobject, metaclass=ConvertToOpenGL):
    def __init__(self,
                 location=ORIGIN,
                 color=BLACK,
                 fill_opacity=0,
                 stroke_width=0,
                 artificial_width=0.01,
                 artificial_height=0.01,
                 **kwargs,
                 ):
        self.artificial_width = artificial_width
        self.artificial_height = artificial_height
        super().__init__(color=color,
                         fill_opacity=fill_opacity,
                         stroke_width=stroke_width,
                         **kwargs,
                         )
        self.set_points(np.array([location]))

    basecls = OpenGLVMobject if config.renderer == RendererType.OPENGL else VMobject

    @basecls.width.getter
    def width(self):
        return self.artificial_width

    @basecls.height.getter
    def height(self):
        return self.artificial_height

    def get_location(self):
        return np.array(self.points[0])

    def set_location(self, new_loc):
        self.set_points(np.array([new_loc]))

class CurvesAsSubmobjects(VGroup):
    """Convert a curve's elements to submobjects.

    Examples
    -----
    .. manim:: LineGradientExample
        :save_last_frame:

        class LineGradientExample(Scene):
            def construct(self):

```

```

curve = ParametricFunction(lambda t: [t, np.sin(t), 0], t_range=[-PI, PI, 0.01], stroke_width=10)
new_curve = CurvesAsSubmobjects(curve)
new_curve.set_color_by_gradient(BLUE, RED)
self.add(new_curve.shift(UP), curve)

"""

def __init__(self, vmobject, **kwargs):
    super().__init__(**kwargs)
    tuples = vmobject.get_cubic_bezier_tuples()
    for tup in tuples:
        part = VMobject()
        part.set_points(tup)
        part.match_style(vmobject)
        self.add(part)

def point_from_proportion(self, alpha: float) -> np.ndarray:
    """Gets the point at a proportion along the path of the :class:`CurvesAsSubmobjects`.

    Parameters
    -----
    alpha
        The proportion along the path of the :class:`CurvesAsSubmobjects`.

    Returns
    -----
    :class:`numpy.ndarray`
        The point on the :class:`CurvesAsSubmobjects`.

    Raises
    -----
    :exc:`ValueError`
        If ``alpha`` is not between 0 and 1.
    :exc:`Exception`
        If the :class:`CurvesAsSubmobjects` has no subobjects, or no subobject has points.
    """
    if alpha < 0 or alpha > 1:
        raise ValueError(f"Alpha {alpha} not between 0 and 1.")

    self._throw_error_if_no_submobjects()
    submobjs_with_pts = self._get_submobjects_with_points()

    if alpha == 1:
        return submobjs_with_pts[-1].points[-1]

    submobjs_arc_lengths = tuple(
        part.get_arc_length() for part in submobjs_with_pts
    )

    total_length = sum(submobjs_arc_lengths)
    target_length = alpha * total_length
    current_length = 0

    for i, part in enumerate(submobjs_with_pts):
        part_length = submobjs_arc_lengths[i]
        if current_length + part_length >= target_length:
            residue = (target_length - current_length) / part_length
            return part.point_from_proportion(residue)

        current_length += part_length

    def _throw_error_if_no_submobjects(self):
        if len(self.submobjects) == 0:
            caller_name = sys._getframe(1).f_code.co_name
            raise Exception(
                f"Cannot call CurvesAsSubmobjects.{caller_name} for a CurvesAsSubmobject with no subobjects"
            )

    def _get_submobjects_with_points(self):
        submobjs_with_pts = tuple(
            part for part in self.submobjects if len(part.points) > 0
        )
        if len(submobjs_with_pts) == 0:
            caller_name = sys._getframe(1).f_code.co_name
            raise Exception(
                f"Cannot call CurvesAsSubmobjects.{caller_name} for a CurvesAsSubmobject whose submobjects have no points"
            )
        return submobjs_with_pts

class DashedVMobject(VMobject, metaclass=ConvertToOpenGL):
    """A :class:`VMobject` composed of dashes instead of lines.

    Parameters
    -----
    vmobject
        The object that will get dashed
    num_dashes
        Number of dashes to add.
    dashed_ratio
        Ratio of dash to empty space.
    dash_offset
        Shifts the starting point of dashes along the path. Value 1 shifts by one full dash length.
    equal_lengths
        If ``True``, dashes will be (approximately) equally long.
        If ``False``, dashes will be split evenly in the curve's input t variable (legacy behavior).

    Examples
    -----
    .. manim:: DashedVMobjectExample
        :save_last_frame:

    class DashedVMobjectExample(Scene):
        def construct(self):
            r = 0.5

            top_row = VGroup() # Increasing num_dashes
            for dashes in range(1, 12):
                circ = DashedVMobject(Circle(radius=r, color=WHITE), num_dashes=dashes)
                top_row.add(circ)

            middle_row = VGroup() # Increasing dashed_ratio
            for ratio in np.arange(1 / 11, 1, 1 / 11):
                circ = DashedVMobject(
                    Circle(radius=r, color=WHITE), dashed_ratio=ratio
                )
                middle_row.add(circ)

            func1 = FunctionGraph(lambda t: t**5, [-1, 1], color=WHITE)
            func_even = DashedVMobject(func1, num_dashes=6, equal_lengths=True)
            func_stretched = DashedVMobject(func1, num_dashes=6, equal_lengths=False)
    """

```

```

bottom_row = VGroup(func_even, func_stretched)

top_row.arrange(buff=0.3)
middle_row.arrange()
bottom_row.arrange(buff=1)
everything = VGroup(top_row, middle_row, bottom_row).arrange(DOWN, buff=1)
self.add(everything)

"""
def __init__(
    self,
    vmobject,
    num_dashes=15,
    dashed_ratio=0.5,
    dash_offset=0,
    color=WHITE,
    equal_lengths=True,
    **kwargs,
):
    self.dashed_ratio = dashed_ratio
    self.num_dashes = num_dashes
    super().__init__(color=color, **kwargs)
    r = self.dashed_ratio
    n = self.num_dashes
    if n > 0:
        # Assuming total length is 1
        dash_len = r / n
        if vmobject.is_closed():
            void_len = (1 - r) / n
        else:
            if n == 1:
                void_len = 1 - r
            else:
                void_len = (1 - r) / (n - 1)

        period = dash_len + void_len
        phase_shift = (dash_offset % 1) * period

        if vmobject.is_closed():
            # closed curves have equal amount of dashes and voids
            pattern_len = 1
        else:
            # open curves start and end with a dash, so the whole dash pattern with the last void is longer
            pattern_len = 1 + void_len

        dash_starts = [((i * period + phase_shift) % pattern_len) for i in range(n)]
        dash_ends = [
            ((i * period + dash_len + phase_shift) % pattern_len) for i in range(n)
        ]

        # closed shapes can handle overflow at the 0-point
        # open shapes need special treatment for it
        if not vmobject.is_closed():
            # due to phase shift being [0...1] range, always the last dash element needs attention for overflow
            # if an entire dash moves out of the shape end:
            if dash_ends[-1] > 1 and dash_starts[-1] > 1:
                # remove the last element since it is out-of-bounds
                dash_ends.pop()
                dash_starts.pop()
            elif dash_ends[-1] < dash_len:  # if it overflowed
                if (
                    dash_starts[-1] < 1
                ):  # if the beginning of the piece is still in range
                    dash_starts.append(0)
                    dash_ends.append(dash_ends[-1])
                    dash_ends[-2] = 1
                else:
                    dash_starts[-1] = 0
            elif dash_starts[-1] > (1 - dash_len):
                dash_ends[-1] = 1

        if equal_lengths:
            # calculate the entire length by adding up short line-pieces
            norms = np.array(0)
            for k in range(vmobject.get_num_curves()):
                norms = np.append(norms, vmobject.get_nth_curve_length_pieces(k))
            # add up length-pieces in array form
            length_vals = np.cumsum(norms)
            ref_points = np.linspace(0, 1, length_vals.size)
            curve_length = length_vals[-1]
            self.add(
                *(
                    vmobject.get_subcurve(
                        np.interp(
                            dash_starts[i] * curve_length,
                            length_vals,
                            ref_points,
                        ),
                        np.interp(
                            dash_ends[i] * curve_length,
                            length_vals,
                            ref_points,
                        ),
                    ),
                    for i in range(len(dash_starts))
                )
            )
        else:
            self.add(
                *(
                    vmobject.get_subcurve(
                        dash_starts[i],
                        dash_ends[i],
                    ),
                    for i in range(len(dash_starts))
                )
            )
    )
# Family is already taken care of by get_subcurve
# implementation
if config.renderer == RendererType.OPENGL:
    self.match_style(vmobject, recurse=False)
else:
    self.match_style(vmobject, family=False)

```

cairo_renderer.py

```
from __future__ import annotations

import typing
from typing import Any

import numpy as np

from manim.utils.hashing import get_hash_from_play_call

from .. import config, logger
from ..camera.camera import Camera
from ..mobject.mobject import Mobject
from ..scene.scene_file_writer import SceneFileWriter
from ..utils.exceptions import EndSceneEarlyException
from ..utils.iterables import list_update

if typing.TYPE_CHECKING:
    from manim.scene.scene import Scene


class CairoRenderer:
    """A renderer using Cairo.

    num_plays : Number of play() functions in the scene.
    time: time elapsed since initialisation of scene.
    """

    def __init__(self,
                 file_writer_class=SceneFileWriter,
                 camera_class=None,
                 skip_animations=False,
                 **kwargs,
                 ):
        # All of the following are set to EITHER the value passed via kwargs,
        # OR the value stored in the global config dict at the time of
        # __instance construction__.
        self._file_writer_class = file_writer_class
        camera_cls = camera_class if camera_class is not None else Camera
        self.camera = camera_cls()
        self._original_skipping_status = skip_animations
        self.skip_animations = skip_animations
        self.animations_hashes = []
        self.num_plays = 0
        self.time = 0
        self.static_image = None

    def init_scene(self, scene):
        self.file_writer: Any = self._file_writer_class(
            self,
            scene.__class__.__name__,
        )

    def play(self, scene, *args, **kwargs):
        # Reset skip_animations to the original state.
        # Needed when rendering only some animations, and skipping others.
        self.skip_animations = self._original_skipping_status
        self.update_skipping_status()

        scene.compile_animation_data(*args, **kwargs)

        if self.skip_animations:
            logger.debug(f"Skipping animation {self.num_plays}")
            hash_current_animation = None
            self.time += scene.duration
        else:
            if config["disable_caching"]:
                logger.info("Caching disabled.")
                hash_current_animation = f"uncached_{self.num_plays:05}"
            else:
                hash_current_animation = get_hash_from_play_call(
                    scene,
                    self.camera,
                    scene.animations,
                    scene.mobjects,
                )
            if self.file_writer.is_already_cached(hash_current_animation):
                logger.info(
                    f"Animation {self.num_plays} : Using cached data (hash : %(hash_current_animation)s)",
                    {"hash_current_animation": hash_current_animation},
                )

```

```

        self.skip_animations = True
        self.time += scene.duration
    # adding None as a partial movie file will make file_writer ignore the latter.
    self.file_writer.add_partial_movie_file(hash_current_animation)
    self.animations_hashes.append(hash_current_animation)
    logger.debug(
        "List of the first few animation hashes of the scene: %(h)s",
        {"h": str(self.animations_hashes[:5])},
    )

    self.file_writer.begin_animation(not self.skip_animations)
    scene.begin_animations()

    # Save a static image, to avoid rendering non moving objects.
    self.save_static_frame_data(scene, scene.static_mobjects)

    if scene.is_current_animation_frozen_frame():
        self.update_frame(scene, mobjects=scene.moving_mobjects)
        # self.duration stands for the total run time of all the animations.
        # In this case, as there is only a wait, it will be the length of the wait.
        self.freeze_current_frame(scene.duration)
    else:
        scene.play_internal()
    self.file_writer.end_animation(not self.skip_animations)

    self.num_plays += 1

def update_frame(  # TODO Description in Docstring
    self,
    scene,
    mobjects: typing.Iterable[Mobject] | None = None,
    include_submobjects: bool = True,
    ignore_skipping: bool = True,
    **kwargs,
):
    """
    Update the frame.

    Parameters
    -----
    scene

    mobjects
        list of mobjects

    include_submobjects

    ignore_skipping

    **kwargs

    """
    if self.skip_animations and not ignore_skipping:
        return
    if not mobjects:
        mobjects = list_update(
            scene.mobjects,
            scene.foreground_mobjects,
        )
    if self.static_image is not None:
        self.camera.set_frame_to_background(self.static_image)
    else:
        self.camera.reset()

    kwargs["include_submobjects"] = include_submobjects
    self.camera.capture_mobjects(mobjects, **kwargs)

def render(self, scene, time, moving_mobjects):
    self.update_frame(scene, moving_mobjects)
    self.add_frame(self.get_frame())

def get_frame(self):
    """
    Gets the current frame as NumPy array.

    Returns
    -----
    np.array
        NumPy array of pixel values of each pixel in screen.
        The shape of the array is height x width x 3
    """
    return np.array(self.camera.pixel_array)

def add_frame(self, frame: np.ndarray, num_frames: int = 1):
    """
    Adds a frame to the video_file_stream

    Parameters

```

```

-----
frame
    The frame to add, as a pixel array.
num_frames
    The number of times to add frame.
"""
dt = 1 / self.camera.frame_rate
if self.skip_animations:
    return
self.time += num_frames * dt
for _ in range(num_frames):
    self.file_writer.write_frame(frame)

def freeze_current_frame(self, duration: float):
    """Adds a static frame to the movie for a given duration. The static frame is the current frame.

Parameters
-----
duration
    [description]
"""
dt = 1 / self.camera.frame_rate
self.add_frame(
    self.get_frame(),
    num_frames=int(duration / dt),
)
)

def show_frame(self):
    """
    Opens the current frame in the Default Image Viewer
    of your system.
    """
    self.update_frame(ignore_skipping=True)
    self.camera.get_image().show()

def save_static_frame_data(
    self,
    scene: Scene,
    static_mobjects: typing.Iterable[Mobject],
) -> typing.Iterable[Mobject] | None:
    """Compute and save the static frame, that will be reused at each frame
    to avoid unnecessarily computing static mobjects.

Parameters
-----
scene
    The scene played.
static_mobjects
    Static mobjects of the scene. If None, self.static_image is set to None

Returns
-----
typing.Iterable[Mobject]
    The static image computed.
"""
self.static_image = None
if not static_mobjects:
    return None
self.update_frame(scene, mobjects=static_mobjects)
self.static_image = self.get_frame()
return self.static_image

def update_skipping_status(self):
    """
    This method is used internally to check if the current
    animation needs to be skipped or not. It also checks if
    the number of animations that were played correspond to
    the number of animations that need to be played, and
    raises an EndSceneEarlyException if they don't correspond.
    """
    # there is always at least one section -> no out of bounds here
    if self.file_writer.sections[-1].skip_animations:
        self.skip_animations = True
    if config["save_last_frame"]:
        self.skip_animations = True
    if (
        config["from_animation_number"]
        and self.num_plays < config["from_animation_number"]
    ):
        self.skip_animations = True
    if (
        config["upto_animation_number"]
        and self.num_plays > config["upto_animation_number"]
    ):
        self.skip_animations = True
        raise EndSceneEarlyException()

```

```
def scene_finished(self, scene):
    # If no animations in scene, render an image instead
    if self.num_plays:
        self.file_writer.finish()
    elif config.write_to_movie:
        config.save_last_frame = True
        config.write_to_movie = False
    else:
        self.static_image = None
        self.update_frame(scene)

    if config["save_last_frame"]:
        self.static_image = None
        self.update_frame(scene)
        self.file_writer.save_final_image(self.camera.get_image())
```

output_options.py

```
from __future__ import annotations

import click
from cloup import option, option_group

output_options = option_group(
    "Output options",
    option(
        "-o",
        "--output_file",
        type=str,
        default=None,
        help="Specify the filename(s) of the rendered scene(s).",
    ),
    option(
        "-0",
        "--zero_pad",
        type=click.IntRange(0, 9),
        default=None,
        help="Zero padding for PNG file names.",
    ),
    option(
        "--write_to_movie",
        is_flag=True,
        default=None,
        help="Write the video rendered with opengl to a file.",
    ),
    option(
        "--media_dir",
        type=click.Path(),
        default=None,
        help="Path to store rendered videos and latex.",
    ),
    option(
        "--log_dir",
        type=click.Path(),
        help="Path to store render logs.",
        default=None,
    ),
    option(
        "--log_to_file",
        is_flag=True,
        default=None,
        help="Log terminal output to file.",
    ),
)
```

iterables.py

```
"""Operations on iterables."""

from __future__ import annotations

__all__ = [
    "adjacent_n_tuples",
    "adjacent_pairs",
    "all_elements_are_instances",
    "concatenate_lists",
    "list_difference_update",
    "list_update",
    "listify",
    "make_even",
    "make_even_by_cycling",
    "remove_list_redundancies",
    "remove_nones",
    "stretch_array_to_length",
    "tuplify",
]
import itertools as it
from typing import Any, Callable, Collection, Generator, Iterable, Reversible, Sequence
import numpy as np

def adjacent_n_tuples(objects: Sequence, n: int) -> zip:
    """Returns the Sequence objects cyclically split into n length tuples.

    See Also
    -----
    adjacent_pairs : alias with n=2

    Examples
    -----
    Normal usage::

        list(adjacent_n_tuples([1, 2, 3, 4], 2))
        # returns [(1, 2), (2, 3), (3, 4), (4, 1)]

        list(adjacent_n_tuples([1, 2, 3, 4], 3))
        # returns [(1, 2, 3), (2, 3, 4), (3, 4, 1), (4, 1, 2)]
    """
    return zip(*([*objects[k:], *objects[:k]] for k in range(n)))

def adjacent_pairs(objects: Sequence) -> zip:
    """Alias for ``adjacent_n_tuples(objects, 2)``.

    See Also
    -----
    adjacent_n_tuples

    Examples
    -----
    Normal usage::

        list(adjacent_pairs([1, 2, 3, 4]))
        # returns [(1, 2), (2, 3), (3, 4), (4, 1)]
    """
    return adjacent_n_tuples(objects, 2)

def all_elements_are_instances(iterable: Iterable, Class) -> bool:
    """Returns ``True`` if all elements of iterable are instances of Class.
    False otherwise.
    """
    return all(isinstance(e, Class) for e in iterable)

def batch_by_property(
    items: Sequence, property_func: Callable
) -> list[tuple[list, Any]]:
    """Takes in a Sequence, and returns a list of tuples, (batch, prop)
    such that all items in a batch have the same output when
    put into the Callable property_func, and such that chaining all these
    batches together would give the original Sequence (i.e. order is
    preserved).

    Examples
    -----
    
```

Normal usage::

```
batch_by_property([(1, 2), (3, 4), (5, 6, 7), (8, 9)], len)
# returns [([(1, 2), (3, 4)], 2), ([(5, 6, 7)], 3), ([(8, 9)], 2)]
"""
batch_prop_pairs = []
curr_batch = []
curr_prop = None
for item in items:
    prop = property_func(item)
    if prop != curr_prop:
        # Add current batch
        if len(curr_batch) > 0:
            batch_prop_pairs.append((curr_batch, curr_prop))
        # Redefine curr
        curr_prop = prop
        curr_batch = [item]
    else:
        curr_batch.append(item)
if len(curr_batch) > 0:
    batch_prop_pairs.append((curr_batch, curr_prop))
return batch_prop_pairs
```

def concatenate_lists(*list_of_lists: Iterable) -> list:
"""Combines the Iterables provided as arguments into one list.

Examples

Normal usage::

```
concatenate_lists([1, 2], [3, 4], [5])
# returns [1, 2, 3, 4, 5]
"""
return [item for lst in list_of_lists for item in lst]
```

def list_difference_update(l1: Iterable, l2: Iterable) -> list:
"""Returns a list containing all the elements of l1 not in l2.

Examples

Normal usage::

```
list_difference_update([1, 2, 3, 4], [2, 4])
# returns [1, 3]
"""
return [e for e in l1 if e not in l2]
```

def list_update(l1: Iterable, l2: Iterable) -> list:
"""Used instead of ``set.update()`` to maintain order,
making sure duplicates are removed from l1, not l2.
Removes overlap of l1 and l2 and then concatenates l2 unchanged.

Examples

Normal usage::

```
list_update([1, 2, 3], [2, 4, 4])
# returns [1, 3, 2, 4, 4]
"""
return [e for e in l1 if e not in l2] + list(l2)
```

def listify(obj) -> list:
"""Converts obj to a list intelligently.

Examples

Normal usage::

```
listify('str')    # ['str']
listify((1, 2))  # [1, 2]
listify(len)     # [built-in function len]
"""
if isinstance(obj, str):
    return [obj]
try:
    return list(obj)
except TypeError:
    return [obj]
```

def make_even(iterator_1: Iterable, iterator_2: Iterable) -> tuple[list, list]:
"""Extends the shorter of the two iterables with duplicate values until its
length is equal to the longer iterable (favours earlier elements).

See Also

make_even_by_cycling : cycles elements instead of favouring earlier ones

Examples

Normal usage::

```
make_even([1, 2], [3, 4, 5, 6])
([1, 1, 2, 2], [3, 4, 5, 6])

make_even([1, 2], [3, 4, 5, 6, 7])
# ([1, 1, 1, 2, 2], [3, 4, 5, 6, 7])
"""
list_1, list_2 = list(iterable_1), list(iterable_2)
len_list_1 = len(list_1)
len_list_2 = len(list_2)
length = max(len_list_1, len_list_2)
return (
    [list_1[(n * len_list_1) // length] for n in range(length)],
    [list_2[(n * len_list_2) // length] for n in range(length)],
)
```



```
def make_even_by_cycling(
    iterable_1: Collection, iterable_2: Collection
) -> tuple[list, list]:
    """Extends the shorter of the two iterables with duplicate values until its
    length is equal to the longer iterable (cycles over shorter iterable).
```

See Also

make_even : favours earlier elements instead of cycling them

Examples

Normal usage::

```
make_even_by_cycling([1, 2], [3, 4, 5, 6])
([1, 2, 1, 2], [3, 4, 5, 6])

make_even_by_cycling([1, 2], [3, 4, 5, 6, 7])
# ([1, 2, 1, 2, 1], [3, 4, 5, 6, 7])
"""
length = max(len(iterable_1), len(iterable_2))
cycle1 = it.cycle(iterable_1)
cycle2 = it.cycle(iterable_2)
return (
    [next(cycle1) for _ in range(length)],
    [next(cycle2) for _ in range(length)],
)
```

```
def remove_list_redundancies(lst: Reversible) -> list:
    """Used instead of ``list(set(l))`` to maintain order.
    Keeps the last occurrence of each element.
    """
    reversed_result = []
    used = set()
    for x in reversed(lst):
        if x not in used:
            reversed_result.append(x)
            used.add(x)
    reversed_result.reverse()
    return reversed_result
```

```
def remove_nones(sequence: Iterable) -> list:
    """Removes elements where bool(x) evaluates to False.
```

Examples

Normal usage::

```
remove_nones(['m', '', 'l', 0, 42, False, True])
# ['m', 'l', 42, True]
"""
# Note this is redundant with it.chain
return [x for x in sequence if x]
```

```
def resize_array(nparray: np.ndarray, length: int) -> np.ndarray:
    """Extends/truncates nparray so that ``len(result) == length``.
    The elements of nparray are cycled to achieve the desired length.
```

See Also

```
-----  
resize_preserving_order : favours earlier elements instead of cycling them  
make_even_by_cycling : similar cycling behaviour for balancing 2 iterables
```

Examples

```
-----
```

```
Normal usage::
```

```
>>> points = np.array([[1, 2], [3, 4]])  
>>> resize_array(points, 1)  
array([[1, 2]])  
>>> resize_array(points, 3)  
array([[1, 2],  
       [3, 4],  
       [1, 2]])  
>>> resize_array(points, 2)  
array([[1, 2],  
       [3, 4]])  
"""  
if len(nparray) == length:  
    return nparray  
return np.resize(nparray, (length, *nparray.shape[1:]))
```

```
def resize_preserving_order(nparray: np.ndarray, length: int) -> np.ndarray:
```

```
"""Extends/truncates nparray so that `len(result) == length``.  
The elements of nparray are duplicated to achieve the desired length  
(favours earlier elements).
```

```
Constructs a zeroes array of length if nparray is empty.
```

See Also

```
-----
```

```
resize_array : cycles elements instead of favouring earlier ones  
make_even : similar earlier-favouring behaviour for balancing 2 iterables
```

Examples

```
-----
```

```
Normal usage::
```

```
resize_preserving_order(np.array([]), 5)  
# np.array([0., 0., 0., 0., 0.])  
  
nparray = np.array([[1, 2],  
                  [3, 4]])  
  
resize_preserving_order(nparray, 1)  
# np.array([[1, 2]])  
  
resize_preserving_order(nparray, 3)  
# np.array([[1, 2],  
#           [1, 2],  
#           [3, 4]])  
"""  
if len(nparray) == 0:  
    return np.zeros((length, *nparray.shape[1:]))  
if len(nparray) == length:  
    return nparray  
indices = np.arange(length) * len(nparray) // length  
return nparray[indices]
```

```
def resize_with_interpolation(nparray: np.ndarray, length: int) -> np.ndarray:
```

```
"""Extends/truncates nparray so that `len(result) == length``.  
New elements are interpolated to achieve the desired length.
```

```
Note that if nparray's length changes, its dtype may too  
(e.g. int -> float: see Examples)
```

See Also

```
-----
```

```
resize_array : cycles elements instead of interpolating
```

```
resize_preserving_order : favours earlier elements instead of interpolating
```

Examples

```
-----
```

```
Normal usage::
```

```
nparray = np.array([[1, 2],  
                  [3, 4]])  
  
resize_with_interpolation(nparray, 1)  
# np.array([[1., 2.]])  
  
resize_with_interpolation(nparray, 4)  
# np.array([[1.          , 2.          ],  
#           [1.66666667, 2.66666667],
```

```

#             [2.33333333, 3.33333333],
#             [3.,           4.        ]])

nparray = np.array([[1, 2], [3, 4]])
resize_with_interpolation(nparray, 3)
# np.array([[1., 2.], [3., 4.]],
#          [[1., 2.], [3., 4.]],
#          [[1., 2.], [3., 4.]]])

nparray = np.array([[1, 2], [3, 4], [5, 6]])
resize_with_interpolation(nparray, 4)
# np.array([1.,           2.        ],
#          [2.33333333, 3.33333333],
#          [3.66666667, 4.66666667],
#          [5.,           6.        ]))

nparray = np.array([[1, 2], [3, 4], [1, 2]])
resize_with_interpolation(nparray, 4)
# np.array([1.,           2.        ],
#          [2.33333333, 3.33333333],
#          [2.33333333, 3.33333333],
#          [1.,           2.        ]])
"""

if len(nparray) == length:
    return nparray
cont_indices = np.linspace(0, len(nparray) - 1, length)
return np.array(
    [
        (1 - a) * nparray[lh] + a * nparray[rh]
        for ci in cont_indices
        for lh, rh, a in [(int(ci), int(np.ceil(ci)), ci % 1)]
    ],
)
)

def stretch_array_to_length(nparray: np.ndarray, length: int) -> np.ndarray:
    # todo: is this the same as resize_preserving_order()?
    curr_len = len(nparray)
    if curr_len > length:
        raise Warning("Trying to stretch array to a length shorter than its own")
    indices = np.arange(length) / float(length)
    indices *= curr_len
    return nparray[indices.astype(int)]

def tuplify(obj) -> tuple:
    """Converts obj to a tuple intelligently.

Examples
-----
Normal usage:::

    tuplify('str')    # ('str',)
    tuplify([1, 2])   # (1, 2)
    tuplify(len)      # (<built-in function len>,)

"""
    if isinstance(obj, str):
        return (obj,)
    try:
        return tuple(obj)
    except TypeError:
        return (obj,)

def uniq_chain(*args: Iterable) -> Generator:
    """Returns a generator that yields all unique elements of the Iterables
    provided via args in the order provided.

Examples
-----
Normal usage:::

    uniq_chain([1, 2], [2, 3], [1, 4, 4])
    # yields 1, 2, 3, 4
"""
    unique_items = set()
    for x in it.chain(*args):
        if x in unique_items:
            continue
        unique_items.add(x)
        yield x

def hash_obj(obj: object) -> int:
    """Determines a hash, even of potentially mutable objects."""
    if isinstance(obj, dict):
        return hash(tuple(sorted((hash_obj(k), hash_obj(v)) for k, v in obj.items())))

```

```
if isinstance(obj, set):
    return hash(tuple(sorted(hash_obj(e) for e in obj)))

if isinstance(obj, (tuple, list)):
    return hash(tuple(hash_obj(e) for e in obj))

return hash(obj)
```

zoomed_scene.py

```
"""A scene supporting zooming in on a specified section.

Examples
-----
.. manim:: UseZoomedScene

class UseZoomedScene(ZoomedScene):
    def construct(self):
        dot = Dot().set_color(GREEN)
        self.add(dot)
        self.wait(1)
        self.activate_zooming(animate=False)
        self.wait(1)
        self.play(dot.animate.shift(LEFT))

.. manim:: ChangingZoomScale

class ChangingZoomScale(ZoomedScene):
    def __init__(self, **kwargs):
        ZoomedScene.__init__(
            self,
            zoom_factor=0.3,
            zoomed_display_height=1,
            zoomed_display_width=3,
            image_frame_stroke_width=20,
            zoomed_camera_config={
                "default_frame_stroke_width": 3,
            },
            **kwargs
        )

    def construct(self):
        dot = Dot().set_color(GREEN)
        sq = Circle(fill_opacity=1, radius=0.2).next_to(dot, RIGHT)
        self.add(dot, sq)
        self.wait(1)
        self.activate_zooming(animate=False)
        self.wait(1)
        self.play(dot.animate.shift(LEFT * 0.3))

        self.play(self.zoomed_camera.frame.animate.scale(4))
        self.play(self.zoomed_camera.frame.animate.shift(0.5 * DOWN))

"""

from __future__ import annotations

__all__ = ["ZoomedScene"]

from ..animation.transform import ApplyMethod
from ..camera.moving_camera import MovingCamera
from ..camera.multi_camera import MultiCamera
from ..constants import *
from ..mobject.types.image_mobject import ImageMobjectFromCamera
from ..scene.moving_camera_scene import MovingCameraScene

# Note, any scenes from old videos using ZoomedScene will almost certainly
# break, as it was restructured.

class ZoomedScene(MovingCameraScene):
    """
    This is a Scene with special configurations made for when
    a particular part of the scene must be zoomed in on and displayed
    separately.
    """

    def __init__(
        self,
        camera_class=MultiCamera,
        zoomed_display_height=3,
        zoomed_display_width=3,
        zoomed_display_center=None,
        zoomed_display_corner=UP + RIGHT,
        zoomed_display_corner_buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER,
        zoomed_camera_config={
            "default_frame_stroke_width": 2,
            "background_opacity": 1,
```

```

},
zoomed_camera_image_mobject_config={},
zoomed_camera_frame_starting_position=ORIGIN,
zoom_factor=0.15,
image_frame_stroke_width=3,
zoom_activated=False,
**kwargs,
):
    self.zoomed_display_height = zoomed_display_height
    self.zoomed_display_width = zoomed_display_width
    self.zoomed_display_center = zoomed_display_center
    self.zoomed_display_corner = zoomed_display_corner
    self.zoomed_display_corner_buff = zoomed_display_corner_buff
    self.zoomed_camera_config = zoomed_camera_config
    self.zoomed_camera_image_mobject_config = zoomed_camera_image_mobject_config
    self.zoomed_camera_frame_starting_position = (
        zoomed_camera_frame_starting_position
    )
    self.zoom_factor = zoom_factor
    self.image_frame_stroke_width = image_frame_stroke_width
    self.zoom_activated = zoom_activated
super().__init__(camera_class=camera_class, **kwargs)

def setup(self):
    """
    This method is used internally by Manim to
    setup the scene for proper use.
    """
    super().setup()
    # Initialize camera and display
    zoomed_camera = MovingCamera(**self.zoomed_camera_config)
    zoomed_display = ImageMobjectFromCamera(
        zoomed_camera, **self.zoomed_camera_image_mobject_config
    )
    zoomed_display.add_display_frame()
    for mob in zoomed_camera.frame, zoomed_display:
        mob.stretch_to_fit_height(self.zoomed_display_height)
        mob.stretch_to_fit_width(self.zoomed_display_width)
    zoomed_camera.frame.scale(self.zoom_factor)

    # Position camera and display
    zoomed_camera.frame.move_to(self.zoomed_camera_frame_starting_position)
    if self.zoomed_display_center is not None:
        zoomed_display.move_to(self.zoomed_display_center)
    else:
        zoomed_display.to_corner(
            self.zoomed_display_corner,
            buff=self.zoomed_display_corner_buff,
        )

    self.zoomed_camera = zoomed_camera
    self.zoomed_display = zoomed_display

def activate_zooming(self, animate: bool = False):
    """
    This method is used to activate the zooming for
    the zoomed_camera.

    Parameters
    -----
    animate
        Whether or not to animate the activation
        of the zoomed camera.
    """
    self.zoom_activated = True
    self.renderer.camera.add_image_mobject_from_camera(self.zoomed_display)
    if animate:
        self.play(self.get_zoom_in_animation())
        self.play(self.get_zoomed_display_pop_out_animation())
    self.add_foreground_mobjects(
        self.zoomed_camera.frame,
        self.zoomed_display,
    )

def get_zoom_in_animation(self, run_time: float = 2, **kwargs):
    """
    Returns the animation of camera zooming in.

    Parameters
    -----
    run_time
        The run_time of the animation of the camera zooming in.
    **kwargs
        Any valid keyword arguments of ApplyMethod()

    Returns
    -----
    """

```

```
ApplyMethod
    The animation of the camera zooming in.
"""
frame = self.zoomed_camera.frame
full_frame_height = self.camera.frame_height
full_frame_width = self.camera.frame_width
frame.save_state()
frame.stretch_to_fit_width(full_frame_width)
frame.stretch_to_fit_height(full_frame_height)
frame.center()
frame.set_stroke(width=0)
return ApplyMethod(frame.restore, run_time=run_time, **kwargs)

def get_zoomed_display_pop_out_animation(self, **kwargs):
"""
This is the animation of the popping out of the
mini-display that shows the content of the zoomed
camera.

Returns
-----
ApplyMethod
    The Animation of the Zoomed Display popping out.
"""
display = self.zoomed_display
display.save_state()
display.replace(self.zoomed_camera.frame, stretch=True)
return ApplyMethod(display.restore)

def get_zoom_factor(self):
"""
Returns the Zoom factor of the Zoomed camera.
Defined as the ratio between the height of the
zoomed camera and the height of the zoomed mini
display.
Returns
-----
float
    The zoom factor.
"""
return self.zoomed_camera.frame.height / self.zoomed_display.height
```

tex_file_writing.py

```
"""Interface for writing, compiling, and converting ``.tex`` files.

.. SEEALSO::

    :mod:`.mobject.svg.tex_mobject`


from __future__ import annotations

import hashlib
import os
import re
import unicodedata
from pathlib import Path

from manim.utils.tex import TexTemplate

from .. import config, logger

def tex_hash(expression):
    id_str = str(expression)
    hasher = hashlib.sha256()
    hasher.update(id_str.encode())
    # Truncating at 16 bytes for cleanliness
    return hasher.hexdigest()[:16]

def tex_to_svg_file(
    expression: str,
    environment: str | None = None,
    tex_template: TexTemplate | None = None,
):
    """Takes a tex expression and returns the svg version of the compiled tex

    Parameters
    -----
    expression
        String containing the TeX expression to be rendered, e.g. ``\\sqrt{2}`` or ``foo``
    environment
        The string containing the environment in which the expression should be typeset, e.g. ``align*``
    tex_template
        Template class used to typesetting. If not set, use default template set via `config["tex_template"]`'

    Returns
    -----
    :class:`Path`
        Path to generated SVG file.
    """
    if tex_template is None:
        tex_template = config["tex_template"]
    tex_file = generate_tex_file(expression, environment, tex_template)
    dvi_file = compile_tex(
        tex_file,
        tex_template.tex_compiler,
        tex_template.output_format,
    )
    return convert_to_svg(dvi_file, tex_template.output_format)

def generate_tex_file(
    expression: str,
    environment: str | None = None,
    tex_template: TexTemplate | None = None,
):
    """Takes a tex expression (and an optional tex environment),
    and returns a fully formed tex file ready for compilation.

    Parameters
    -----
    expression
        String containing the TeX expression to be rendered, e.g. ``\\sqrt{2}`` or ``foo``
    environment
        The string containing the environment in which the expression should be typeset, e.g. ``align*``
    tex_template
        Template class used to typesetting. If not set, use default template set via `config["tex_template"]`'

    Returns
    -----
    :class:`Path`
        Path to generated TeX file
    """

```

```

if tex_template is None:
    tex_template = config["tex_template"]
if environment is not None:
    output = tex_template.get_texcode_for_expression_in_env(expression, environment)
else:
    output = tex_template.get_texcode_for_expression(expression)

tex_dir = config.get_dir("tex_dir")
if not tex_dir.exists():
    tex_dir.mkdir()

result = tex_dir / (tex_hash(output) + ".tex")
if not result.exists():
    logger.info(
        "Writing %(expression)s to %(path)s",
        {"expression": expression, "path": f"{result}"},
    )
    result.write_text(output, encoding="utf-8")
return result

def tex_compilation_command(
    tex_compiler: str, output_format: str, tex_file: Path, tex_dir: Path
) -> str:
    """Prepares the tex compilation command with all necessary cli flags

    Parameters
    -----
    tex_compiler
        String containing the compiler to be used, e.g. ``pdflatex`` or ``lualatex``
    output_format
        String containing the output format generated by the compiler, e.g. ``.dvi`` or ``.pdf``
    tex_file
        File name of TeX file to be typeset.
    tex_dir
        Path to the directory where compiler output will be stored.

    Returns
    -----
    :class:`str`
        Compilation command according to given parameters
    """
    if tex_compiler in {"latex", "pdflatex", "luatex", "lualatex"}:
        commands = [
            tex_compiler,
            "-interaction=batchmode",
            f"--output-format='{output_format[1:]}'",
            "-halt-on-error",
            f"--output-directory='{tex_dir.as_posix()}'",
            f'"{tex_file.as_posix()}"',
            ">",
            os.devnull,
        ]
    elif tex_compiler == "xelatex":
        if output_format == ".xdv":
            outflag = "-no-pdf"
        elif output_format == ".pdf":
            outflag = ""
        else:
            raise ValueError("xelatex output is either pdf or xdv")
        commands = [
            "xelatex",
            outflag,
            "-interaction=batchmode",
            "-halt-on-error",
            f"--output-directory='{tex_dir.as_posix()}'",
            f'"{tex_file.as_posix()}"',
            ">",
            os.devnull,
        ]
    else:
        raise ValueError(f"Tex compiler {tex_compiler} unknown.")
    return " ".join(commands)

def insight_inputenc_error(matching):
    code_point = chr(int(matching[1], 16))
    name = unicodedata.name(code_point)
    yield f"TexTemplate does not support character '{name}' (U+{matching[1]})."
    yield "See the documentation for manim.mobject.svg.tex_mobject for details on using a custom TexTemplate."

def insight_package_not_found_error(matching):
    yield f"You do not have package {matching[1]} installed."
    yield f"Install {matching[1]} it using your LaTeX package manager, or check for typos."

def compile_tex(tex_file: Path, tex_compiler: str, output_format: str) -> Path:
    """Compiles a tex_file into a .dvi or a .xdv or a .pdf

```

```

Parameters
-----
tex_file
    File name of TeX file to be typeset.
tex_compiler
    String containing the compiler to be used, e.g. ``pdflatex`` or ``lualatex``.
output_format
    String containing the output format generated by the compiler, e.g. ``.dvi`` or ``.pdf``

Returns
-----
:class:`Path`
    Path to generated output file in desired format (DVI, XDV or PDF).
"""
result = tex_file.with_suffix(output_format)
tex_dir = config.get_dir("tex_dir")
if not result.exists():
    command = tex_compilation_command(
        tex_compiler,
        output_format,
        tex_file,
        tex_dir,
    )
    exit_code = os.system(command)
    if exit_code != 0:
        log_file = tex_file.with_suffix(".log")
        print_all_tex_errors(log_file, tex_compiler, tex_file)
        raise ValueError(
            f"{tex_compiler} error converting to"
            f" {output_format[1:]}. See log output above or"
            f" the log file: {log_file}",
        )
return result

```

```

def convert_to_svg(dvi_file: Path, extension: str, page: int = 1):
    """Converts a .dvi, .xdv, or .pdf file into an svg using dvisvgm.

```

```

Parameters
-----
dvi_file
    File name of the input file to be converted.
extension
    String containing the file extension and thus indicating the file type, e.g. ``.dvi`` or ``.pdf``.
page
    Page to be converted if input file is multi-page.

Returns
-----
:class:`Path`
    Path to generated SVG file.
"""
result = dvi_file.with_suffix(".svg")
if not result.exists():
    commands = [
        "dvisvgm",
        "--pdf" if extension == ".pdf" else "",
        "-p " + str(page),
        f'{dvi_file.as_posix()}',
        "-n",
        "-v 0",
        "-o " + f'{result.as_posix()}',
        ">",
        os.devnull,
    ]
    os.system(" ".join(commands))

# if the file does not exist now, this means conversion failed
if not result.exists():
    raise ValueError(
        f"Your installation does not support converting {dvi_file.suffix} files to SVG."
        f" Consider updating dvisvgm to at least version 2.4."
        f" If this does not solve the problem, please refer to our troubleshooting guide at:"
        f" https://docs.manim.community/en/stable/faq/general.html#my-installation-"
        f"does-not-support-converting-pdf-to-svg-help",
    )
return result

```

```

def print_all_tex_errors(log_file: Path, tex_compiler: str, tex_file: Path) -> None:
    if not log_file.exists():
        raise RuntimeError(
            f"{tex_compiler} failed but did not produce a log file. "
            "Check your LaTeX installation.",
        )
    with log_file.open(encoding="utf-8") as f:
        tex_compilation_log = f.readlines()

```

```

error_indices = [
    index for index, line in enumerate(tex_compilation_log) if line.startswith("!")
]
if error_indices:
    with tex_file.open() as f:
        tex = f.readlines()
    for error_index in error_indices:
        print_tex_error(tex_compilation_log, error_index, tex)

LATEX_ERROR_INSIGHTS = [
(
    r"inputenc Error: Unicode character (?:.* ) \U\+([0-9a-fA-F]+)\ )",
    insight_inputenc_error,
),
(
    r"LaTeX Error: File `(.?*[clsty])' not found",
    insight_package_not_found_error,
),
]
]

def print_tex_error(tex_compilation_log, error_start_index, tex_source):
    logger.error(
        f"LaTeX compilation error: {tex_compilation_log[error_start_index][2:]}",

    )

    # TeX errors eventually contain a line beginning 'l.xxx` where xxx is the line number that caused the compilation
    # failure. This code finds the next such line after the error current error message
    line_of_tex_error = (
        int(
            [
                log_line
                for log_line in tex_compilation_log[error_start_index:]
                if log_line.startswith("l.")
            ][0]
            .split(" ")[0]
            .split(".") [1],
        )
        - 1
    )
    # our tex error may be on a line outside our user input because of post-processing
    if line_of_tex_error >= len(tex_source):
        return None

    context = ["Context of error: \n"]
    if line_of_tex_error < 3:
        context += tex_source[: line_of_tex_error + 3]
        context[-4] = "-> " + context[-4]
    elif line_of_tex_error > len(tex_source) - 3:
        context += tex_source[line_of_tex_error - 1 : ]
        context[1] = "-> " + context[1]
    else:
        context += tex_source[line_of_tex_error - 3 : line_of_tex_error + 3]
        context[-4] = "-> " + context[-4]

    context = "".join(context)
    logger.error(context)

    for insights in LATEX_ERROR_INSIGHTS:
        prob, get_insight = insights
        matching = re.search(
            prob,
            "".join(tex_compilation_log[error_start_index])[2:],
        )
        if matching is not None:
            for insight in get_insight(matching):
                logger.info(insight)

```

tex_mobject.py

```
r"""Mobjects representing text rendered using LaTeX.

.. important::

    See the corresponding tutorial :ref:`rendering-with-latex`


.. note::

    Just as you can use :class:`~.Text` (from the module :mod:`~.text_mobject`) to add text to your videos, you can use :class:`~.Tex` and :class:`~.MathTex` to insert LaTeX.

"""

from __future__ import annotations

__all__ = [
    "SingleStringMathTex",
    "MathTex",
    "Tex",
    "BulletedList",
    "Title",
]

import itertools as it
import operator as op
import re
from functools import reduce
from textwrap import dedent
from typing import Dict, Iterable, Optional

from colour import Color

from manim import config, logger
from manim.constants import *
from manim.mobject.geometry.line import Line
from manim.mobject.svg.svg_mobject import SVGObject
from manim.mobject.types.vectorized_mobject import VectorizedPoint, VGroup, VMobject
from manim.utils.tex import TexTemplate
from manim.utils.tex_file_writing import tex_to_svg_file

SCALE_FACTOR_PER_FONT_POINT = 1 / 960

tex_string_to_mob_map = {}

class SingleStringMathTex(SVGObject):
    """Elementary building block for rendering text with LaTeX.

    Tests
    -----
    Check that creating a :class:`~.SingleStringMathTex` object works:

    >>> SingleStringMathTex('Test') # doctest: +SKIP
    SingleStringMathTex('Test')
    """

    def __init__(self,
                 tex_string: str,
                 stroke_width: float = 0,
                 should_center: bool = True,
                 height: float | None = None,
                 organize_left_to_right: bool = False,
                 tex_environment: str = "align*",
                 tex_template: TexTemplate | None = None,
                 font_size: float = DEFAULT_FONT_SIZE,
                 **kwargs,
                 ):
        if kwargs.get("color") is None:
            # makes it so that color isn't explicitly passed for these mobs,
            # and can instead inherit from the parent
            kwargs["color"] = VMobject().color

        self._font_size = font_size
        self.organize_left_to_right = organize_left_to_right
        self.tex_environment = tex_environment
        if tex_template is None:
            tex_template = config["tex_template"]
        self.tex_template = tex_template

        assert isinstance(tex_string, str)
        self.tex_string = tex_string
        file_name = tex_to_svg_file(
            self._get_modified_expression(tex_string),
            environment=self.tex_environment,
            tex_template=self.tex_template,
        )
        super().__init__(
            file_name=file_name,
            should_center=should_center,
            stroke_width=stroke_width,
            height=height,
            path_string_config={
                "should_subdivide_sharp_curves": True,
                "should_remove_null_curves": True,
            },
            **kwargs,
        )
        self.init_colors()

        # used for scaling via font_size.setter
        self.initial_height = self.height

        if height is None:
            self.font_size = self._font_size

        if self.organize_left_to_right:
            self._organize_submobjects_left_to_right()

    def __repr__(self):
        return f"{type(self).__name__}({repr(self.tex_string)})"

    @property
    def font_size(self):
        """The font size of the tex mobject."""
        return self.height / self.initial_height / SCALE_FACTOR_PER_FONT_POINT

    @font_size.setter
    def font_size(self, font_val):
        if font_val < 0:
            raise ValueError("font_size must be greater than 0.")
        elif self.height > 0:
            # sometimes manim generates a SingleStringMathTex mobject with 0 height.
            # can't be scaled regardless and will error without the elif.
            # scale to a factor of the initial height so that setting

```

```

    # font_size does not depend on current size.
    self.scale(font_val / self.font_size)

def _get_modified_expression(self, tex_string):
    result = tex_string
    result = result.strip()
    result = self._modify_special_strings(result)
    return result

def _modify_special_strings(self, tex):
    tex = tex.strip()
    should_add_filler = reduce(
        op.or_,
        [
            # Fraction line needs something to be over
            tex == "\overline",
            tex == "\overline{",
            # Make sure sqrt has overbar
            tex == "\sqrt",
            tex == "\sqrt{",
            # Need to add blank subscript or superscript
            tex.endswith("."),
            tex.endswith("^{"),
            tex.endswith("_{"),
            tex.endswith("dot"),
        ],
    )

    if should_add_filler:
        filler = "\quad"
        tex += filler

    if tex == "\substack":
        tex = "\quad"

    if tex == "":
        tex = "\quad"

    # To keep files from starting with a line break
    if tex.startswith("\\"):
        tex = tex.replace("\\", "\quad\\\"")

    # Handle imbalanced \left and \right
    num_lefts, num_rights = (
        len([s for s in tex.split(substr)[1:] if s and s[0] in "()[]\\"]
            for substr in ("\\left", "\\right"))
    )
    if num_lefts != num_rights:
        tex = tex.replace("\\left", "\\big")
        tex = tex.replace("\\right", "\\big")

    tex = self._remove_stray_braces(tex)

for context in ["array"]:
    begin_in = ("\\begin{\\s}" % context) in tex
    end_in = ("\\end{\\s}" % context) in tex
    if begin_in ^ end_in:
        # Just turn this into a blank string,
        # which means caller should leave a
        # stray \\begin{...} with other symbols
        tex = ""
    return tex

def _remove_stray_braces(self, tex):
    """
    Makes :class:`~.MathTex` resilient to unmatched braces.

    This is important when the braces in the TeX code are spread over
    multiple arguments as in, e.g., ``MathTex(r"e^{i", r"\tau} = 1")``.
    """
    # "(" does not count (it's a brace literal), but "\\{" counts (it's a new line and then brace)
    num_lefts = tex.count("(") - tex.count("\\(") + tex.count("\\\\{")
    num_rights = tex.count(")") - tex.count("\\)") + tex.count("\\\\\\")
    while num_rights > num_lefts:
        tex = "(" + tex
        num_lefts += 1
    while num_lefts > num_rights:
        tex = tex + ")"
        num_rights += 1
    return tex

def _organize_subobjects_left_to_right(self):
    self.sort(lambda p: p[0])
    return self

def get_tex_string(self):
    return self.tex_string

def init_colors(self, propagate_colors=True):
    if config.renderer == RendererType.OPENGL:
        super().init_colors()
    elif config.renderer == RendererType.CAIRO:
        super().init_colors(propagate_colors=propagate_colors)

class MathTex(SingleStringMathTex):
    """
    A string compiled with LaTeX in math mode.

    Examples
    ======
    ... manim:: Formula
    :save_last_frame:

    class Formula(Scene):
        def construct(self):
            t = MathTex(r"\int_a^b f'(x) dx = f(b) - f(a)")
            self.add(t)

    Tests
    -----
    Check that creating a :class:`~.MathTex` works::

        >>> MathTex('a^2 + b^2 = c^2') # doctest: +SKIP
        MathTex('a^2 + b^2 = c^2')

    Check that double brace group splitting works correctly::

        >>> t1 = MathTex('{{ a }} + {{ b }} = {{ c }}') # doctest: +SKIP
        >>> len(t1.submobjects) # doctest: +SKIP
        5
        >>> t2 = MathTex(r"\frac{1}{a+b}\sqrt{2})") # doctest: +SKIP
        >>> len(t2.submobjects) # doctest: +SKIP
        1
    """
    def __init__(
        self,
        *tex_strings,
        arg_separator: str = " ",
        substrings_to_isolate: Iterable[str] | None = None,
        tex_to_color_map: dict[str, Color] = None,
    ):

```

```

tex_environment: str = "align*",  

**kwargs,  
):  
    self.tex_template = kwargs.pop("tex_template", config["tex_template"])  
    self.arg_separator = arg_separator  
    self.substrings_to_isolate = (  
        [] if substrings_to_isolate is None else substrings_to_isolate  
    )  
    self.tex_to_color_map = tex_to_color_map  
    if self.tex_to_color_map is None:  
        self.tex_to_color_map = {}  
    self.tex_environment = tex_environment  
    self.brace_notation_split_occurred = False  
    self.tex_strings = self._break_up_tex_strings(tex_strings)  
    try:  
        super().__init__(  
            self.arg_separator.join(self.tex_strings),  
            tex_environment=tex_environment,  
            tex_template=self.tex_template,  
            **kwargs,  
        )  
        self._break_up_by_substrings()  
    except ValueError as compilation_error:  
        if self.brace_notation_split_occurred:  
            logger.error(  
                dedent(  
                    """  
                        A group of double braces, {{ ... }}, was detected in  
                        your string. Manim splits TeX strings at the double  
                        braces, which might have caused the current  
                        compilation error. If you didn't use the double brace  
                        split intentionally, add spaces between the braces to  
                        avoid the automatic splitting: {{ ... }} --> { { ... } }.  
                    """,  
                ),  
            )  
            raise compilation_error  
    self.set_color_by_tex_to_color_map(self.tex_to_color_map)  
  
if self.organize_left_to_right:  
    self._organize_submobjects_left_to_right()  
  
def _break_up_tex_strings(self, tex_strings):  
    # Separate out anything surrounded in double braces  
    pre_split_length = len(tex_strings)  
    tex_strings = [re.split("{{(.*)}}", str(t)) for t in tex_strings]  
    tex_strings = sum(tex_strings, [])  
    if len(tex_strings) > pre_split_length:  
        self.brace_notation_split_occurred = True  
  
    # Separate out any strings specified in the isolate  
    # or tex_to_color_map lists.  
    patterns = []  
    patterns.extend(  
        [  
            f"({re.escape(ss)})"  
            for ss in it.chain(  
                self.substrings_to_isolate,  
                self.tex_to_color_map.keys(),  
            )  
        ],  
    )  
    pattern = "|".join(patterns)  
    if pattern:  
        pieces = []  
        for s in tex_strings:  
            pieces.extend(re.split(pattern, s))  
    else:  
        pieces = tex_strings  
    return [p for p in pieces if p]  
  
def _break_up_by_substrings(self):  
    """  
    Reorganize existing submobjects one layer  
    deeper based on the structure of tex_strings (as a list  
    of tex_strings)  
    """  
    new_submobjects = []  
    curr_index = 0  
    for tex_string in self.tex_strings:  
        sub_tex_mob = SingleStringMathTex(  
            tex_string,  
            tex_environment=self.tex_environment,  
            tex_template=self.tex_template,  
        )  
        num_submobs = len(sub_tex_mob.submobjects)  
        new_index = (  
            curr_index + num_submobs + len("".join(self.arg_separator.split())))
        )  
        if num_submobs == 0:  
            # For cases like empty tex_strings, we want the corresponding  
            # part of the whole MathTex to be a VectorizedPoint  
            # positioned in the right part of the MathTex  
            sub_tex_mob.submobjects = [VectorizedPoint()]  
            last_submob_index = min(curr_index, len(self.submobjects) - 1)  
            sub_tex_mob.move_to(self.submobjects[last_submob_index], RIGHT)
        else:  
            sub_tex_mob.submobjects = self.submobjects[curr_index:new_index]  
            new_submobjects.append(sub_tex_mob)
        curr_index = new_index  
    self.submobjects = new_submobjects  
    return self  
  
def get_parts_by_tex(self, tex, substring=True, case_sensitive=True):  
    def test(tex1, tex2):  
        if not case_sensitive:  
            tex1 = tex1.lower()  
            tex2 = tex2.lower()
        if substring:  
            return tex1 in tex2
        else:  
            return tex1 == tex2
    return VGroup(*[m for m in self.submobjects if test(tex, m.get_tex_string())])  
  
def get_part_by_tex(self, tex, **kwargs):  
    all_parts = self.get_parts_by_tex(tex, **kwargs)
    return all_parts[0] if all_parts else None  
  
def set_color_by_tex(self, tex, color, **kwargs):  
    parts_to_color = self.get_parts_by_tex(tex, **kwargs)
    for part in parts_to_color:  
        part.set_color(color)
    return self  
  
def set_opacity_by_tex(  
    self, tex: str, opacity: float = 0.5, remaining_opacity: float = None, **kwargs
):  
    """  
    Sets the opacity of the tex specified. If 'remaining_opacity' is specified,  
    then the remaining tex will be set to that opacity.

```

```

Parameters
-----
tex
    The tex to set the opacity of.
opacity
    Default 0.5. The opacity to set the tex to
remaining_opacity
    Default None. The opacity to set the remaining tex to.
    If None, then the remaining tex will not be changed
"""
if remaining_opacity is not None:
    self.set_opacity(opacity=remaining_opacity)
for part in self.get_parts_by_tex(tex):
    part.set_opacity(opacity)
return self

def set_color_by_tex_to_color_map(self, texts_to_color_map, **kwargs):
    for texts, color in list(texts_to_color_map.items()):
        try:
            # If the given key behaves like tex_strings
            texts + ""
            self.set_color_by_tex(texts, color, **kwargs)
        except TypeError:
            # If the given key is a tuple
            for tex in texts:
                self.set_color_by_tex(tex, color, **kwargs)
    return self

def index_of_part(self, part):
    split_self = self.split()
    if part not in split_self:
        raise ValueError("Trying to get index of part not in MathTex")
    return split_self.index(part)

def index_of_part_by_tex(self, tex, **kwargs):
    part = self.get_part_by_tex(tex, **kwargs)
    return self.index_of_part(part)

def sort_alphabetically(self):
    self.submobjects.sort(key=lambda m: m.get_tex_string())

class Tex(MathTex):
    """
    A string compiled with LaTeX in normal mode.

    Tests
    -----
    Check whether writing a LaTeX string works::

        >>> Tex('The horse does not eat cucumber salad.') # doctest: +SKIP
        Tex('The horse does not eat cucumber salad.')

    """

    def __init__(self, *tex_strings, arg_separator="", tex_environment="center", **kwargs):
        super().__init__(
            *tex_strings,
            arg_separator=arg_separator,
            tex_environment=tex_environment,
            **kwargs,
        )

class BulletedList(Tex):
    """
    A bulleted list.

    Examples
    -----
    .. manim:: BulletedListExample
       :save_last_frame:

    class BulletedListExample(Scene):
        def construct(self):
            blist = BulletedList("Item 1", "Item 2", "Item 3", height=2, width=2)
            blist.set_color_by_tex("Item 1", RED)
            blist.set_color_by_tex("Item 2", GREEN)
            blist.set_color_by_tex("Item 3", BLUE)
            self.add(blist)

    """

    def __init__(self,
                 items,
                 buff=MED_LARGE_BUFF,
                 dot_scale_factor=2,
                 tex_environment=None,
                 **kwargs,
                 ):
        self.buff = buff
        self.dot_scale_factor = dot_scale_factor
        self.tex_environment = tex_environment
        line_separated_items = [s + "\\\\" for s in items]
        super().__init__(
            *line_separated_items, tex_environment=tex_environment, **kwargs
        )
        for part in self:
            dot = MathTex("\cdot").scale(self.dot_scale_factor)
            dot.next_to(part[0], LEFT, SMALL_BUFF)
            part.add_back(dot)
        self.arrange(DOWN, aligned_edge=LEFT, buff=self.buff)

    def fade_all_but(self, index_or_string, opacity=0.5):
        arg = index_or_string
        if isinstance(arg, str):
            part = self.get_part_by_tex(arg)
        elif isinstance(arg, int):
            part = self.submobjects[arg]
        else:
            raise TypeError(f"Expected int or string, got {arg}")
        for other_part in self.submobjects:
            if other_part is part:
                other_part.set_fill(opacity=1)
            else:
                other_part.set_fill(opacity=opacity)

class Title(Tex):
    """
    A mobject representing an underlined title.

    Examples
    -----
    .. manim:: TitleExample
       :save_last_frame:

       import manim

       class TitleExample(Scene):
           def construct(self):

```

```
banner = ManimBanner()
title = Title(f"Manim version {manim.__version__}")
self.add(banner, title)

"""

def __init__(
    self,
    *text_parts,
    include_underline=True,
    match_underline_width_to_text=False,
    underline_buff=MED_SMALL_BUFF,
    **kwargs,
):
    self.include_underline = include_underline
    self.match_underline_width_to_text = match_underline_width_to_text
    self.underline_buff = underline_buff
    super().__init__(*text_parts, **kwargs)
    self.to_edge(DP)
    if self.include_underline:
        underline_width = config["frame_width"] - 2
        underline = Line(LEFT, RIGHT)
        underline.next_to(self, DOWN, buff=self.underline_buff)
        if self.match_underline_width_to_text:
            underline.match_width(self)
        else:
            underline.width = underline_width
        self.add(underline)
        self.underline = underline
```

shader_wrapper.py

```
from __future__ import annotations

import copy
import re
from pathlib import Path

import moderngl
import numpy as np

from .. import logger

# Mobjects that should be rendered with
# the same shader will be organized and
# clumped together based on keeping track
# of a dict holding all the relevant information
# to that shader

def get_shader_dir():
    return Path(__file__).parent / "shaders"

def find_file(file_name: Path, directories: list[Path]) -> Path:
    # Check if what was passed in is already a valid path to a file
    if file_name.exists():
        return file_name
    possible_paths = (directory / file_name for directory in directories)
    for path in possible_paths:
        if path.exists():
            return path
        else:
            logger.debug(f"{path} does not exist.")
    raise OSError(f"{file_name} not Found")

class ShaderWrapper:
    def __init__(
        self,
        vert_data=None,
        vert_indices=None,
        shader_folder=None,
        uniforms=None, # A dictionary mapping names of uniform variables
        texture_paths=None, # A dictionary mapping names to filepaths for textures.
        depth_test=False,
        render_primitive=moderngl.TRIANGLE_STRIP,
    ):
        self.vert_data = vert_data
        self.vert_indices = vert_indices
        self.vert_attributes = vert_data.dtype.names
        self.shader_folder = Path(shader_folder or "")
        self.uniforms = uniforms or {}
        self.texture_paths = texture_paths or {}
        self.depth_test = depth_test
        self.render_primitive = str(render_primitive)
        self.init_program_code()
        self.refresh_id()

    def copy(self):
        result = copy.copy(self)
        result.vert_data = np.array(self.vert_data)
        if result.vert_indices is not None:
            result.vert_indices = np.array(self.vert_indices)
        if self.uniforms:
            result.uniforms = dict(self.uniforms)
        if self.texture_paths:
            result.texture_paths = dict(self.texture_paths)
        return result

    def is_valid(self):
        return all(
            [
                self.vert_data is not None,
                self.program_code["vertex_shader"] is not None,
                self.program_code["fragment_shader"] is not None,
            ],
        )

    def get_id(self):
        return self.id

    def get_program_id(self):
```

```

    return self.program_id

def create_id(self):
    # A unique id for a shader
    return "|".join(
        map(
            str,
            [
                self.program_id,
                self.uniforms,
                self.texture_paths,
                self.depth_test,
                self.render_primitive,
            ],
        ),
    )

def refresh_id(self):
    self.program_id = self.create_program_id()
    self.id = self.create_id()

def create_program_id(self):
    return hash(
        "".join(
            self.program_code[f"{name}_shader"] or ""
            for name in ("vertex", "geometry", "fragment")
        ),
    )

def init_program_code(self):
    def get_code(name: str) -> str | None:
        return get_shader_code_from_file(
            self.shader_folder / f"{name}.glsl",
        )

    self.program_code = {
        "vertex_shader": get_code("vert"),
        "geometry_shader": get_code("geom"),
        "fragment_shader": get_code("frag"),
    }

def get_program_code(self):
    return self.program_code

def replace_code(self, old, new):
    code_map = self.program_code
    for name, _code in code_map.items():
        if code_map[name] is None:
            continue
        code_map[name] = re.sub(old, new, code_map[name])
    self.refresh_id()

def combine_with(self, *shader_wrappers):
    # Assume they are of the same type
    if len(shader_wrappers) == 0:
        return
    if self.vert_indices is not None:
        num_verts = len(self.vert_data)
        indices_list = [self.vert_indices]
        data_list = [self.vert_data]
        for sw in shader_wrappers:
            indices_list.append(sw.vert_indices + num_verts)
            data_list.append(sw.vert_data)
            num_verts += len(sw.vert_data)
        self.vert_indices = np.hstack(indices_list)
        self.vert_data = np.hstack(data_list)
    else:
        self.vert_data = np.hstack(
            [self.vert_data, *(sw.vert_data for sw in shader_wrappers)],
        )
    return self

# For caching
filename_to_code_map: dict = {}

def get_shader_code_from_file(filename: Path) -> str | None:
    if filename in filename_to_code_map:
        return filename_to_code_map[filename]

    try:
        filepath = find_file(
            filename,
            directories=[get_shader_dir(), Path("/")],
        )
    except OSError:

```

```
    return None

result = filepath.read_text()

# To share functionality between shaders, some functions are read in
# from other files and inserted into the relevant strings before
# passing to ctx.program for compiling
# Replace "#INSERT " lines with relevant code
insertions = re.findall(
    r"^\#include ../include/.*\.glsl$",
    result,
    flags=re.MULTILINE,
)
for line in insertions:
    inserted_code = get_shader_code_from_file(
        Path() / "include" / line.replace("#include ../include/", ""),
    )
    if inserted_code is None:
        return None
    result = result.replace(line, inserted_code)
filename_to_code_map[filename] = result
return result
```

```
def get_colormap_code(rgb_list):
    data = ",".join("vec3({}, {}, {})".format(*rgb) for rgb in rgb_list)
    return f"vec3[{len(rgb_list)}]({{data}})"
```

```

# manim.cfg
# Default configuration for manim

# Configure how manim behaves when called from the command line without
# specifying any flags
[CLI]

# Each of the following will be set to True if the corresponding CLI flag
# is present when executing manim. If the flag is not present, they will
# be set to the value found here. For example, running manim with the -w
# flag will set WRITE_TO_MOVIE to True. However, since the default value
# of WRITE_TO_MOVIE defined in this file is also True, running manim
# without the -w value will also output a movie file. To change that, set
# WRITE_TO_MOVIE = False so that running manim without the -w flag will not
# generate a movie file. Note all of the following accept boolean values.

# --notify_outdated_version
notify_outdated_version = True

# -w, --write_to_movie
write_to_movie = True

format = mp4

# -s, --save_last_frame
# setting save_last_frame to True forces write_to_movie to False
save_last_frame = False

# -a, --write_all
write_all = False

# -g, --save_pngs
save_pngs = False

# -0, --zero_pad
zero_pad = 4

# -i, --save_as_gif
save_as_gif = False

# --save_sections
save_sections = False

# -p, --preview
preview = False

# -f, --show_in_file_browser
show_in_file_browser = False

# -v, --verbosity
verbosity = INFO

# --progress_bar
progress_bar = display

# -o, --output_file
output_file =

# --log_to_file
log_to_file = False

# -c, --background_color
background_color = BLACK

# --background_opacity
background_opacity = 1

# The following two are both set by the -n (or --from_animation_number)
# flag. See manim -h for more information.
from_animation_number = 0

# Use -1 to render all animations
upto_animation_number = -1

# The following are meant to be paths relative to the point of execution.
# Set them at the CLI with the corresponding flag, or set their default
# values here.

# --media_dir
media_dir = ./media

# --log_dir
log_dir = {media_dir}/logs

# --assets_dir
assets_dir = ./
```

```

# the following do not have CLI arguments but depend on media_dir
video_dir = {media_dir}/videos/{module_name}/{quality}
sections_dir = {video_dir}/sections
images_dir = {media_dir}/images/{module_name}
tex_dir = {media_dir}/Tex
text_dir = {media_dir}/texts
partial_movie_dir = {video_dir}/partial_movie_files/{scene_name}

# --renderer [cairo|opengl]
renderer = cairo

# --enable_gui
enable_gui = False

# --gui_location
gui_location = 0,0

# --fullscreen
fullscreen = False

# "Set the position of preview window. You can use directions, e.g. UL/DR/LEFT/ORIGIN
# or the position (pixel) of the upper left corner of the window, e.g. '960,540'",
# --window_position
window_position = UR

# Manually adjust the size of the window, or use default to automatically scale the window based on
# the dimensions of the monitor.
# --window_size
window_size = default

# --window_monitor
window_monitor = 0

# --force_window
force_window = False

# --use_projection_fill_shaders
use_projection_fill_shaders = False

# --use_projection_stroke_shaders
use_projection_stroke_shaders = False

movie_file_extension = .mp4

# These now override the --quality option.
frame_rate = 60
pixel_height = 1080
pixel_width = 1920

# Use -1 to set max_files_cached to infinity.
max_files_cached = 100
#Flush cache will delete all the cached partial-movie-files.
flush_cache = False
disable_caching = False
# Disable the warning when there are too much subobjects to hash.
disable_caching_warning = False

# --enable_wireframe
enable_wireframe = False

# --dry_run
dry_run = False

# Default tex_template
# --tex_template
tex_template =

# specify the plugins as comma separated values
# manim will load that plugin if it specified here.
plugins =

# CLI Context/Formatter
# Visit the cloup documentation to understand the formatting options available:
# https://cloup.readthedocs.io/en/latest/index.html#a-simple-example
[CLI_CTX]
# CTX settings
align_option_groups = True
align_sections = True
show_constraints = True

# Formatter settings
indent_increment = 2
width = 80
coll_max_width = 30
col2_min_width = 35
col_spacing = 2
row_sep =

```

```
# Dark/Light, or leave empty
theme = "Dark"

# Theme Settings - The following options override the theme colors.
command_help =
invoked_command =
heading =
constraint =
section_help =
col1 =
col2 =
epilog =

# Overrides the default output folders, NOT the output file names. Note that
# if the custom_folders flag is present, the Tex and text files will not be put
# under media_dir, as is the default.
[custom_folders]
media_dir = videos
video_dir = {media_dir}
sections_dir = {media_dir}
images_dir = {media_dir}
text_dir = {media_dir}/temp_files
tex_dir = {media_dir}/temp_files
log_dir = {media_dir}/temp_files
partial_movie_dir = {media_dir}/partial_movie_files/{scene_name}

# Rich settings
[logger]
logging_keyword = bold yellow
logging_level_notset = dim
logging_level_debug = green
logging_level_info = green
logging_level_warning = red
logging_level_error = red bold
logging_level_critical = red bold reverse
log_level =
log_time = cyan dim
log_message =
log_path = dim
log_width = -1
log_height = -1
log_timestamps = True
repr_number = green

[ffmpeg]
# Uncomment the following line to manually set the loglevel for ffmpeg. See
# ffmpeg manpage for accepted values
loglevel = ERROR
# defaults to the one present in path
ffmpeg_executable = ffmpeg

[jupyter]
media_embed =
media_width = 60%%
```

global_options.py

```
from __future__ import annotations

import re
from typing import TYPE_CHECKING

import click
from cloup import option, option_group

from ... import logger

if TYPE_CHECKING:
    from cloup._option_groups import OptionGroupDecorator


def validate_gui_location(ctx, param, value):
    if value:
        try:
            x_offset, y_offset = map(int, re.split(r"[;,\-]", value))
            return (x_offset, y_offset)
        except Exception:
            logger.error("GUI location option is invalid.")
            exit()

global_options: OptionGroupDecorator = option_group(
    "Global options",
    option(
        "-c",
        "--config_file",
        help="Specify the configuration file to use for render settings.",
        default=None,
    ),
    option(
        "--custom_folders",
        is_flag=True,
        default=None,
        help="Use the folders defined in the [custom_folders] section of the "
        "config file to define the output folder structure.",
    ),
    option(
        "--disable_caching",
        is_flag=True,
        default=None,
        help="Disable the use of the cache (still generates cache files.).",
    ),
    option(
        "--flush_cache",
        is_flag=True,
        help="Remove cached partial movie files.",
        default=None,
    ),
    option("--tex_template", help="Specify a custom TeX template file.", default=None),
    option(
        "-v",
        "--verbosity",
        type=click.Choice(
            ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"],
            case_sensitive=False,
        ),
        help="Verbosity of CLI output. Changes ffmpeg log level unless 5+.",
        default=None,
    ),
    option(
        "--notify_outdated_version/--silent",
        is_flag=True,
        default=None,
        help="Display warnings for outdated installation.",
    ),
    option(
        "--enable_gui",
        is_flag=True,
        help="Enable GUI interaction.",
        default=None,
    ),
    option(
        "--gui_location",
        default=None,
        callback=validate_gui_location,
        help="Starting location for the GUI.",
    ),
    option(
        "--fullscreen",
        is_flag=True,
        help="Expand the window to its maximum possible size.",
        default=None,
    ),
)
```

```
),
option(
    "--enable_wireframe",
    is_flag=True,
    help="Enable wireframe debugging mode in opengl.",
    default=None,
),
option(
    "--force_window",
    is_flag=True,
    help="Force window to open when using the opengl renderer, intended for debugging as it may impact performance",
    default=False,
),
option(
    "--dry_run",
    is_flag=True,
    help="Renders animations without outputting image or video files and disables the window",
    default=False,
),
)
```

mapping_camera.py

```

"""A camera that allows mapping between objects."""
from __future__ import annotations

__all__ = ["MappingCamera", "OldMultiCamera", "SplitScreenCamera"]

import math

import numpy as np

from ..camera.camera import Camera
from ..mobject.types.vectorized_mobject import VMobject
from ..utils.config_ops import DictAsObject

# TODO: Add an attribute to mobjects under which they can specify that they should just
# map their centers but remain otherwise undistorted (useful for labels, etc.)

class MappingCamera(Camera):
    """Camera object that allows mapping
    between objects.
    """
    def __init__(self,
                 mapping_func=lambda p: p,
                 min_num_curves=50,
                 allow_object_intrusion=False,
                 **kwargs,
                 ):
        self.mapping_func = mapping_func
        self.min_num_curves = min_num_curves
        self.allow_object_intrusion = allow_object_intrusion
        super().__init__(**kwargs)

    def points_to_pixel_coords(self, mobject, points):
        return super().points_to_pixel_coords(
            mobject,
            np.apply_along_axis(self.mapping_func, 1, points),
        )

    def capture_mobjects(self, mobjects, **kwargs):
        mobjects = self.get_mobjects_to_display(mobjects, **kwargs)
        if self.allow_object_intrusion:
            mobject_copies = mobjects
        else:
            mobject_copies = [mobject.copy() for mobject in mobjects]
        for mobject in mobject_copies:
            if (
                isinstance(mobject, VMobject)
                and 0 < mobject.get_num_curves() < self.min_num_curves
            ):
                mobject.insert_n_curves(self.min_num_curves)
        super().capture_mobjects(
            mobject_copies,
            include_submobjects=False,
            excluded_mobjects=None,
        )

    # Note: This allows layering of multiple cameras onto the same portion of the pixel array,
    # the later cameras overwriting the former
    #
    # TODO: Add optional separator borders between cameras (or perhaps peel this off into a
    # CameraPlusOverlay class)

    # TODO, the classes below should likely be deleted
class OldMultiCamera(Camera):
    def __init__(self, *cameras_with_start_positions, **kwargs):
        self.shifted_cameras = [
            DictAsObject(
                {
                    "camera": camera_with_start_positions[0],
                    "start_x": camera_with_start_positions[1][1],
                    "start_y": camera_with_start_positions[1][0],
                    "end_x": camera_with_start_positions[1][1]
                    + camera_with_start_positions[0].pixel_width,
                    "end_y": camera_with_start_positions[1][0]
                    + camera_with_start_positions[0].pixel_height,
                },
            ),
        ]

```

```

        )
        for camera_with_start_positions in cameras_with_start_positions
    ]
super().__init__(**kwargs)

def capture_mobjects(self, mobjects, **kwargs):
    for shifted_camera in self.shifted_cameras:
        shifted_camera.camera.capture_mobjects(mobjects, **kwargs)

    self.pixel_array[
        shifted_camera.start_y : shifted_camera.end_y,
        shifted_camera.start_x : shifted_camera.end_x,
    ] = shifted_camera.camera.pixel_array

def set_background(self, pixel_array, **kwargs):
    for shifted_camera in self.shifted_cameras:
        shifted_camera.camera.set_background(
            pixel_array[
                shifted_camera.start_y : shifted_camera.end_y,
                shifted_camera.start_x : shifted_camera.end_x,
            ],
            **kwargs,
        )

def set_pixel_array(self, pixel_array, **kwargs):
    super().set_pixel_array(pixel_array, **kwargs)
    for shifted_camera in self.shifted_cameras:
        shifted_camera.camera.set_pixel_array(
            pixel_array[
                shifted_camera.start_y : shifted_camera.end_y,
                shifted_camera.start_x : shifted_camera.end_x,
            ],
            **kwargs,
        )

def init_background(self):
    super().init_background()
    for shifted_camera in self.shifted_cameras:
        shifted_camera.camera.init_background()

# A OldMultiCamera which, when called with two full-size cameras, initializes itself
# as a split screen, also taking care to resize each individual camera within it

class SplitScreenCamera(OldMultiCamera):
    def __init__(self, left_camera, right_camera, **kwargs):
        Camera.__init__(self, **kwargs) # to set attributes such as pixel_width
        self.left_camera = left_camera
        self.right_camera = right_camera

        half_width = math.ceil(self.pixel_width / 2)
        for camera in [self.left_camera, self.right_camera]:
            camera.reset_pixel_shape(camera.pixel_height, half_width)

        super().__init__(
            (left_camera, (0, 0)),
            (right_camera, (0, half_width)),
        )

```



```

@lru_cache(maxsize=10)
def choose(n: int, k: int) -> int:
    r"""The binomial coefficient n choose k.

    :math:`\binom{n}{k}` describes the number of possible choices of
    :math:`k` elements from a set of :math:`n` elements.

    References
    -----
    - https://en.wikipedia.org/wiki/Combination
    - https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.comb.html
    """
    return special.comb(n, k, exact=True)

def clip(a, min_a, max_a):
    """Clips ``a`` to the interval [``min_a``, ``max_a``].

    Accepts any comparable objects (i.e. those that support <, >).
    Returns ``a`` if it is between ``min_a`` and ``max_a``.
    Otherwise, whichever of ``min_a`` and ``max_a`` is closest.

    Examples
    -----
    ::

        >>> clip(15, 11, 20)
        15
        >>> clip('a', 'h', 'k')
        'h'
    """
    if a < min_a:
        return min_a
    elif a > max_a:
        return max_a
    return a

def get_parameters(function: Callable) -> MappingProxyType[str, inspect.Parameter]:
    """Return the parameters of ``function`` as an ordered mapping of parameters' names to their corresponding ``Parameter`` objects.

    Examples
    -----
    ::

        >>> get_parameters(get_parameters)
        mappingproxy(OrderedDict([('function', <Parameter "function: 'Callable'">)]))

        >>> tuple(get_parameters(choose))
        ('n', 'k')
    """
    return inspect.signature(function).parameters

def sigmoid(x: float) -> float:
    r"""Returns the output of the logistic function.

    The logistic function, a common example of a sigmoid function, is defined as :math:`\frac{1}{1 + e^{-x}}`.

    References
    -----
    - https://en.wikipedia.org/wiki/Sigmoid\_function
    - https://en.wikipedia.org/wiki/Logistic\_function
    """
    return 1.0 / (1 + np.exp(-x))

```

image_mobject.py

```
"""Mobjects representing raster images."""

from __future__ import annotations

__all__ = ["AbstractImageMobject", "ImageMobject", "ImageMobjectFromCamera"]

import pathlib

import colour
import numpy as np
from PIL import Image
from PIL.Image import Resampling

from manim.mobject.geometry.shape_matchers import SurroundingRectangle

from ... import config
from ...constants import *
from ...mobject.mobject import Mobject
from ...utils.bezier import interpolate
from ...utils.color import WHITE, color_to_int_rgb
from ...utils.images import change_to_rgba_array, get_full_raster_image_path

class AbstractImageMobject(Mobject):
    """
    Automatically filters out black pixels

    Parameters
    -----
    scale_to_resolution
        At this resolution the image is placed pixel by pixel onto the screen, so it
        will look the sharpest and best.
        This is a custom parameter of ImageMobject so that rendering a scene with
        e.g. the ``--quality low`` or ``--quality medium`` flag for faster rendering
        won't effect the position of the image on the screen.
    """

    def __init__(self,
                 scale_to_resolution: int,
                 pixel_array_dtype="uint8",
                 resampling_algorithm=Resampling.BICUBIC,
                 **kwargs,
                 ):
        self.pixel_array_dtype = pixel_array_dtype
        self.scale_to_resolution = scale_to_resolution
        self.set_resampling_algorithm(resampling_algorithm)
        super().__init__(**kwargs)

    def get_pixel_array(self):
        raise NotImplementedError()

    def set_color(self, color, alpha=None, family=True):
        # Likely to be implemented in subclasses, but no obligation
        pass

    def set_resampling_algorithm(self, resampling_algorithm: int):
        """
        Sets the interpolation method for upscaling the image. By default the image is
        interpolated using bicubic algorithm. This method lets you change it.
        Interpolation is done internally using Pillow, and the function besides the
        string constants describing the algorithm accepts the Pillow integer constants.

        Parameters
        -----
        resampling_algorithm
            An integer constant described in the Pillow library,
            or one from the RESAMPLING_ALGORITHMS global dictionary,
            under the following keys:

            * 'bicubic' or 'cubic'
            * 'nearest' or 'none'
            * 'box'
            * 'bilinear' or 'linear'
            * 'hamming'
            * 'lanczos' or 'antialias'
        """

        if isinstance(resampling_algorithm, int):
            self.resampling_algorithm = resampling_algorithm
        else:
            raise ValueError(
```

```

    "resampling_algorithm has to be an int, one of the values defined in "
    "RESAMPLING_ALGORITHMS or a Pillow resampling filter constant. "
    "Available algorithms: 'bicubic', 'nearest', 'box', 'bilinear', "
    "'hamming', 'lanczos'.",
)

def reset_points(self):
    # Corresponding corners of image are fixed to these 3 points
    self.points = np.array(
        [
            UP + LEFT,
            UP + RIGHT,
            DOWN + LEFT,
        ],
    )
    self.center()
    h, w = self.get_pixel_array().shape[:2]
    if self.scale_to_resolution:
        height = h / self.scale_to_resolution * config["frame_height"]
    else:
        height = 3 # this is the case for ImageMobjectFromCamera
    self.stretch_to_fit_height(height)
    self.stretch_to_fit_width(height * w / h)

class ImageMobject(AbstractImageMobject):
    """Displays an Image from a numpy array or a file.

Parameters
-----
scale_to_resolution
    At this resolution the image is placed pixel by pixel onto the screen, so it
    will look the sharpest and best.
    This is a custom parameter of ImageMobject so that rendering a scene with
    e.g. the ``--quality low`` or ``--quality medium`` flag for faster rendering
    won't effect the position of the image on the screen.

Example
-----
.. manim:: ImageFromArray
:save_last_frame:

class ImageFromArray(Scene):
    def construct(self):
        image = ImageMobject(np.uint8([[0, 100, 30, 200],
                                       [255, 0, 5, 33]]))
        image.height = 7
        self.add(image)

Changing interpolation style:
.. manim:: ImageInterpolationEx
:save_last_frame:

class ImageInterpolationEx(Scene):
    def construct(self):
        img = ImageMobject(np.uint8([[63, 0, 0, 0],
                                    [0, 127, 0, 0],
                                    [0, 0, 191, 0],
                                    [0, 0, 0, 255]
                                   ]))

        img.height = 2
        img1 = img.copy()
        img2 = img.copy()
        img3 = img.copy()
        img4 = img.copy()
        img5 = img.copy()

        img1.set_resampling_algorithm(RESAMPLING_ALGORITHMS["nearest"])
        img2.set_resampling_algorithm(RESAMPLING_ALGORITHMS["lanczos"])
        img3.set_resampling_algorithm(RESAMPLING_ALGORITHMS["linear"])
        img4.set_resampling_algorithm(RESAMPLING_ALGORITHMS["cubic"])
        img5.set_resampling_algorithm(RESAMPLING_ALGORITHMS["box"])

        img1.add(Text("nearest").scale(0.5).next_to(img1,UP))
        img2.add(Text("lanczos").scale(0.5).next_to(img2,UP))
        img3.add(Text("linear").scale(0.5).next_to(img3,UP))
        img4.add(Text("cubic").scale(0.5).next_to(img4,UP))
        img5.add(Text("box").scale(0.5).next_to(img5,UP))

        x= Group(img1,img2,img3,img4,img5)
        x.arrange()
        self.add(x)
"""

```

```

def __init__(  

    self,  

    filename_or_array,  

    scale_to_resolution: int = QUALITIES[DEFAULT_QUALITY]["pixel_height"],  

    invert=False,  

    image_mode="RGBA",  

    **kwargs,  
):  

    self.fill_opacity = 1  

    self.stroke_opacity = 1  

    self.invert = invert  

    self.image_mode = image_mode  

    if isinstance(filename_or_array, (str, pathlib.PurePath)):  

        path = get_full_raster_image_path(filename_or_array)  

        image = Image.open(path).convert(self.image_mode)  

        self.pixel_array = np.array(image)  

        self.path = path  

    else:  

        self.pixel_array = np.array(filename_or_array)  

    self.pixel_array_dtype = kwargs.get("pixel_array_dtype", "uint8")  

    self.pixel_array = change_to_rgba_array(  

        self.pixel_array, self.pixel_array_dtype  

    )  

    if self.invert:  

        self.pixel_array[:, :, :3] = 255 - self.pixel_array[:, :, :3]  

super().__init__(scale_to_resolution, **kwargs)

def get_pixel_array(self):  

    """A simple getter method."""  

    return self.pixel_array

def set_color(self, color, alpha=None, family=True):  

    rgb = color_to_int_rgb(color)  

    self.pixel_array[:, :, :3] = rgb  

    if alpha is not None:  

        self.pixel_array[:, :, 3] = int(255 * alpha)  

    for submob in self.submobjects:  

        submob.set_color(color, alpha, family)  

    self.color = color  

    return self

def set_opacity(self, alpha: float):  

    """Sets the image's opacity.  

  

    Parameters  

    -----  

    alpha  

        The alpha value of the object, 1 being opaque and 0 being  

        transparent.  

    """  

    self.pixel_array[:, :, 3] = int(255 * alpha)  

    self.fill_opacity = alpha  

    self.stroke_opacity = alpha  

    return self

def fade(self, darkness: float = 0.5, family: bool = True):  

    """Sets the image's opacity using a 1 - alpha relationship.  

  

    Parameters  

    -----  

    darkness  

        The alpha value of the object, 1 being transparent and 0 being  

        opaque.  

    family  

        Whether the submobjects of the ImageMobject should be affected.  

    """  

    self.set_opacity(1 - darkness)  

    super().fade(darkness, family)  

    return self

def interpolate_color(  

    self, mobobject1: ImageMobject, mobobject2: ImageMobject, alpha: float  
):  

    """Interpolates the array of pixel color values from one ImageMobject  

    into an array of equal size in the target ImageMobject.  

  

    Parameters  

    -----  

    mobobject1  

        The ImageMobject to transform from.  

  

    mobobject2  

        The ImageMobject to transform into.  

  

    alpha  

        Used to track the lerp relationship. Not opacity related.  

    """
```

```

        assert mobject1.pixel_array.shape == mobject2.pixel_array.shape, (
            f"Mobject pixel array shapes incompatible for interpolation.\n"
            f"Mobject 1 ({mobject1}) : {mobject1.pixel_array.shape}\n"
            f"Mobject 2 ({mobject2}) : {mobject2.pixel_array.shape}"
        )
        self.fill_opacity = interpolate(
            mobject1.fill_opacity,
            mobject2.fill_opacity,
            alpha,
        )
        self.stroke_opacity = interpolate(
            mobject1.stroke_opacity,
            mobject2.stroke_opacity,
            alpha,
        )
        self.pixel_array = interpolate(
            mobject1.pixel_array,
            mobject2.pixel_array,
            alpha,
        ).astype(self.pixel_array_dtype)

    def get_style(self):
        return {
            "fill_color": colour.rgb2hex(self.color.get_rgb()),
            "fill_opacity": self.fill_opacity,
        }

    # TODO, add the ability to have the dimensions/orientation of this
    # mobject more strongly tied to the frame of the camera it contains,
    # in the case where that's a MovingCamera

    class ImageMobjectFromCamera(AbstractImageMobject):
        def __init__(self, camera, default_display_frame_config=None, **kwargs):
            self.camera = camera
            if default_display_frame_config is None:
                default_display_frame_config = {
                    "stroke_width": 3,
                    "stroke_color": WHITE,
                    "buff": 0,
                }
            self.default_display_frame_config = default_display_frame_config
            self.pixel_array = self.camera.pixel_array
            super().__init__(scale_to_resolution=False, **kwargs)

        # TODO: Get rid of this.
        def get_pixel_array(self):
            self.pixel_array = self.camera.pixel_array
            return self.pixel_array

        def add_display_frame(self, **kwargs):
            config = dict(self.default_display_frame_config)
            config.update(kwargs)
            self.display_frame = SurroundingRectangle(self, **config)
            self.add(self.display_frame)
            return self

        def interpolate_color(self, mobject1, mobject2, alpha):
            assert mobject1.pixel_array.shape == mobject2.pixel_array.shape, (
                f"Mobject pixel array shapes incompatible for interpolation.\n"
                f"Mobject 1 ({mobject1}) : {mobject1.pixel_array.shape}\n"
                f"Mobject 2 ({mobject2}) : {mobject2.pixel_array.shape}"
            )
            self.pixel_array = interpolate(
                mobject1.pixel_array,
                mobject2.pixel_array,
                alpha,
            ).astype(self.pixel_array_dtype)

```

sounds.py

```
"""Sound-related utility functions."""

from __future__ import annotations

__all__ = [
    "get_full_sound_file_path",
]

from .. import config
from ..utils.file_ops import seek_full_path_from_defaults

# Still in use by add_sound() function in scene_file_writer.py
def get_full_sound_file_path(sound_file_name):
    return seek_full_path_from_defaults(
        sound_file_name,
        default_dir=config.get_dir("assets_dir"),
        extensions=[".wav", ".mp3"],
    )
```

plugins_flags.py

```
"""Plugin Managing Utility"""

from __future__ import annotations

import pkg_resources

from manim import console

__all__ = ["list_plugins"]

def get_plugins():
    plugins = {
        entry_point.name: entry_point.load()
        for entry_point in pkg_resources.iter_entry_points("manim.plugins")
    }
    return plugins

def list_plugins():
    console.print("[green bold]Plugins:[/green bold]", justify="left")

    plugins = get_plugins()
    for plugin in plugins:
        console.print(f"  {plugin}")
```

line.py

```
r"""Mobjects that are lines or variations of them."""
from __future__ import annotations

__all__ = [
    "Line",
    "DashedLine",
    "TangentLine",
    "Elbow",
    "Arrow",
    "Vector",
    "DoubleArrow",
    "Angle",
    "RightAngle",
]
from typing import Any, Sequence

import numpy as np
from colour import Color

from manim import config
from manim.constants import *
from manim.mobject.geometry.arc import Arc, ArcBetweenPoints, Dot, TipableVMobject
from manim.mobject.geometry.tips import ArrowTriangleFilledTip
from manim.mobject.mobject import Mobject
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl.opengl_mobject import OpenGLOBJECT
from manim.mobject.types.vectorized_mobject import DashedVMobject, VGroup, VMobject
from manim.utils.color import *
from manim.utils.color import Colors
from manim.utils.space_ops import angle_of_vector, line_intersection, normalize

class Line(TipableVMobject):
    def __init__(self, start=LEFT, end=RIGHT, buff=0, path_arc=None, **kwargs):
        self.dim = 3
        self.buff = buff
        self.path_arc = path_arc
        self._set_start_and_end_attrs(start, end)
        super().__init__(**kwargs)

    def generate_points(self):
        self.set_points_by_ends(
            start=self.start,
            end=self.end,
            buff=self.buff,
            path_arc=self.path_arc,
        )

    def set_points_by_ends(self, start, end, buff=0, path_arc=0):
        if path_arc:
            arc = ArcBetweenPoints(self.start, self.end, angle=self.path_arc)
            self.set_points(arc.points)
        else:
            self.set_points_as_corners([start, end])

        self._account_for_buff(buff)

    init_points = generate_points

    def _account_for_buff(self, buff):
        if buff == 0:
            return
        #
        if self.path_arc == 0:
            length = self.get_length()
        else:
            length = self.get_arc_length()
        #
        if length < 2 * buff:
            return
        buff_proportion = buff / length
        self.pointwise_become_partial(self, buff_proportion, 1 - buff_proportion)
        return self

    def _set_start_and_end_attrs(self, start, end):
        # If either start or end are Mobjects, this
        # gives their centers
        rough_start = self._pointify(start)
        rough_end = self._pointify(end)
        vect = normalize(rough_end - rough_start)
        # Now that we know the direction between them,
        # we can find the appropriate boundary point from
        # start and end, if they're mobjects
        self.start = self._pointify(start, vect)
        self.end = self._pointify(end, -vect)

    def _pointify(
        self,
        mob_or_point: Mobject | Sequence[float],
        direction: Sequence[float] | None = None,
    ) -> np.ndarray:
        """Transforms a mobject into its corresponding point. Does nothing if a point is passed.

        ``direction`` determines the location of the point along its bounding box in that direction.

        Parameters
        -----
        mob_or_point
            The mobject or point.
        direction
            The direction.
        """
        if isinstance(mob_or_point, (Mobject, OpenGLOBJECT)):
            mob = mob_or_point
            if direction is None:
                return mob.get_center()
            else:
                return mob.get_boundary_point(direction)
        return np.array(mob_or_point)
```

```

def set_path_arc(self, new_value):
    self.path_arc = new_value
    self.init_points()

def put_start_and_end_on(self, start: Sequence[float], end: Sequence[float]):
    """Sets starts and end coordinates of a line.

Examples
-----
.. manim:: LineExample

    class LineExample(Scene):
        def construct(self):
            d = VGroup()
            for i in range(0,10):
                d.add(Dot())
            d.arrange_in_grid(buff=1)
            self.add(d)
            l= Line(d[0], d[1])
            self.add(l)
            self.wait()
            l.put_start_and_end_on(d[1].get_center(), d[2].get_center())
            self.wait()
            l.put_start_and_end_on(d[4].get_center(), d[7].get_center())
            self.wait()

    """
    curr_start, curr_end = self.get_start_and_end()
    if np.all(curr_start == curr_end):
        # TODO, any problems with resetting
        # these attrs?
        self.start = start
        self.end = end
        self.generate_points()
    return super().put_start_and_end_on(start, end)

def get_vector(self):
    return self.get_end() - self.get_start()

def get_unit_vector(self):
    return normalize(self.get_vector())

def get_angle(self):
    return angle_of_vector(self.get_vector())

def get_projection(self, point: Sequence[float]) -> Sequence[float]:
    """Returns the projection of a point onto a line.

Parameters
-----
point
    The point to which the line is projected.

    """
    start = self.get_start()
    end = self.get_end()
    unit_vect = normalize(end - start)
    return start + np.dot(point - start, unit_vect) * unit_vect

def get_slope(self):
    return np.tan(self.get_angle())

def set_angle(self, angle, about_point=None):
    if about_point is None:
        about_point = self.get_start()

    self.rotate(
        angle - self.get_angle(),
        about_point=about_point,
    )

    return self

def set_length(self, length):
    return self.scale(length / self.get_length())

class DashedLine(Line):
    """A dashed :class:`Line`.

Parameters
-----
args
    Arguments to be passed to :class:`Line`
dash_length
    The length of each individual dash of the line.
dashed_ratio
    The ratio of dash space to empty space. Range of 0-1.
kwargs
    Additional arguments to be passed to :class:`Line`

.. seealso::
    :class:`~.DashedVMobject`

Examples
-----
.. manim:: DashedLineExample
    :save_last_frame:

    class DashedLineExample(Scene):
        def construct(self):
            # dash_length increased
            dashed_1 = DashedLine(config.left_side, config.right_side, dash_length=2.0).shift(UP*2)
            # normal
            dashed_2 = DashedLine(config.left_side, config.right_side)
            # dashed_ratio decreased
            dashed_3 = DashedLine(config.left_side, config.right_side, dashed_ratio=0.1).shift(DOWN*2)
            self.add(dashed_1, dashed_2, dashed_3)

    """
    def __init__(self,
                 *args: Any,
                 dash_length: float = DEFAULT_DASH_LENGTH,
                 dashed_ratio: float = 0.5,
                 **kwargs,
                 ):
        self.dash_length = dash_length

```

```

self.dashed_ratio = dashed_ratio
super().__init__(*args, **kwargs)
dashes = DashedVMobject(
    self,
    num_dashes=self._calculate_num_dashes(),
    dashed_ratio=dashed_ratio,
)
self.clear_points()
self.add(*dashes)

def _calculate_num_dashes(self) -> int:
    """Returns the number of dashes in the dashed line.

Examples
-----
:::

>>> DashedLine().__calculate_num_dashes()
20
"""

# Minimum number of dashes has to be 2
return max(
    2,
    int(np.ceil((self.get_length() / self.dash_length) * self.dashed_ratio)),
)

def get_start(self) -> np.ndarray:
    """Returns the start point of the line.

Examples
-----
:::

>>> DashedLine().get_start()
array([-1.,  0.,  0.])
"""

if len(self.subobjects) > 0:
    return self.subobjects[0].get_start()
else:
    return super().get_start()

def get_end(self) -> np.ndarray:
    """Returns the end point of the line.

Examples
-----
:::

>>> DashedLine().get_end()
array([1., 0., 0.])
"""

if len(self.subobjects) > 0:
    return self.subobjects[-1].get_end()
else:
    return super().get_end()

def get_first_handle(self) -> np.ndarray:
    """Returns the point of the first handle.

Examples
-----
:::

>>> DashedLine().get_first_handle()
array([-0.98333333,  0.          ,  0.          ])
"""

return self.subobjects[0].points[1]

def get_last_handle(self) -> np.ndarray:
    """Returns the point of the last handle.

Examples
-----
:::

>>> DashedLine().get_last_handle()
array([0.98333333,  0.          ,  0.          ])
"""

return self.subobjects[-1].points[-2]

class TangentLine(Line):
    """Constructs a line tangent to a :class:`~.VMobject` at a specific point.

Parameters
-----
vmob
    The VMobject on which the tangent line is drawn.
alpha
    How far along the shape that the line will be constructed. range: 0-1.
length
    Length of the tangent line.
d_alpha
    The ``dx`` value
kwargs
    Additional arguments to be passed to :class:`Line`


.. seealso::
    :meth:`~.VMobject.point_from_proportion`


Examples
-----
.. manim:: TangentLineExample
    :save_last_frame:

    class TangentLineExample(Scene):
        def construct(self):
            circle = Circle(radius=2)
            line_1 = TangentLine(circle, alpha=0.0, length=4, color=BLUE_D) # right
            line_2 = TangentLine(circle, alpha=0.4, length=4, color=GREEN) # top left
            self.add(circle, line_1, line_2)
"""

```

```

def __init__(self, vmob: VMobject, alpha: float, length: float = 1, d_alpha: float = 1e-6, **kwargs):
    self.length = length
    self.d_alpha = d_alpha
    da = self.d_alpha
    a1 = np.clip(alpha - da, 0, 1)
    a2 = np.clip(alpha + da, 0, 1)
    super().__init__(vmob.point_from_proportion(a1), vmob.point_from_proportion(a2), **kwargs)
    self.scale(self.length / self.get_length())

class Elbow(VMobject, metaclass=ConvertToOpenGL):
    """Two lines that create a right angle about each other: L-shape.

    Parameters
    -----
    width
        The length of the elbow's sides.
    angle
        The rotation of the elbow.
    kwargs
        Additional arguments to be passed to :class:`~.VMobject`.

    .. seealso::
        :class:`RightAngle`"""

    Examples
    -----
    .. manim:: ElbowExample
        :save_last_frame:

    class ElbowExample(Scene):
        def construct(self):
            elbow_1 = Elbow()
            elbow_2 = Elbow(width=2.0)
            elbow_3 = Elbow(width=2.0, angle=5*PI/4)

            elbow_group = Group(elbow_1, elbow_2, elbow_3).arrange(buff=1)
            self.add(elbow_group)
    """

    def __init__(self, width: float = 0.2, angle: float = 0, **kwargs):
        self.angle = angle
        super().__init__(**kwargs)
        self.set_points_as_corners([UP, UP + RIGHT, RIGHT])
        self.scale_to_fit_width(width, about_point=ORIGIN)
        self.rotate(self.angle, about_point=ORIGIN)

class Arrow(Line):
    """An arrow.

    Parameters
    -----
    args
        Arguments to be passed to :class:`Line`.
    stroke_width
        The thickness of the arrow. Influenced by :attr:`max_stroke_width_to_length_ratio`.
    buff
        The distance of the arrow from its start and end points.
    max_tip_length_to_length_ratio
        :attr:`tip_length` scales with the length of the arrow. Increasing this ratio raises the max value of :attr:`tip_length`.
    max_stroke_width_to_length_ratio
        :attr:`stroke_width` scales with the length of the arrow. Increasing this ratio ratios the max value of :attr:`stroke_width`.
    kwargs
        Additional arguments to be passed to :class:`Line`.

    .. seealso::
        :class:`ArrowTip`
        :class:`CurvedArrow`"""

    Examples
    -----
    .. manim:: ArrowExample
        :save_last_frame:

    from manim.mobject.geometry.tips import ArrowSquareTip
    class ArrowExample(Scene):
        def construct(self):
            arrow_1 = Arrow(start=RIGHT, end=LEFT, color=GOLD)
            arrow_2 = Arrow(start=RIGHT, end=LEFT, color=GOLD, tip_shape=ArrowSquareTip).shift(DOWN)
            g1 = Group(arrow_1, arrow_2)

            # the effect of buff
            square = Square(color=MAROON_A)
            arrow_3 = Arrow(start=LEFT, end=RIGHT)
            arrow_4 = Arrow(start=LEFT, end=RIGHT, buff=0).next_to(arrow_1, UP)
            g2 = Group(arrow_3, arrow_4, square)

            # a shorter arrow has a shorter tip and smaller stroke width
            arrow_5 = Arrow(start=ORIGIN, end=config.top).shift(LEFT * 4)
            arrow_6 = Arrow(start=config.top + DOWN, end=config.top).shift(LEFT * 3)
            g3 = Group(arrow_5, arrow_6)

            self.add(Group(g1, g2, g3).arrange(buff=2))

    .. manim:: ArrowExample
        :save_last_frame:

    class ArrowExample(Scene):
        def construct(self):
            left_group = VGroup()
            # As buff increases, the size of the arrow decreases.
            for buff in np.arange(0, 2.2, 0.45):
                left_group += Arrow(buff=buff, start=2 * LEFT, end=2 * RIGHT)
            # Required to arrange arrows.
            left_group.arrange(DOWN)

```

```

    left_group.move_to(4 * LEFT)

    middle_group = VGroup()
    # As max_stroke_width_to_length_ratio gets bigger,
    # the width of stroke increases.
    for i in np.arange(0, 5, 0.5):
        middle_group += Arrow(max_stroke_width_to_length_ratio=i)
    middle_group.arrange(DOWN)

    UR_group = VGroup()
    # As max_tip_length_to_length_ratio increases,
    # the length of the tip increases.
    for i in np.arange(0, 0.3, 0.1):
        UR_group += Arrow(max_tip_length_to_length_ratio=i)
    UR_group.arrange(DOWN)
    UR_group.move_to(4 * RIGHT + 2 * UP)

    DR_group = VGroup()
    DR_group += Arrow(start=LEFT, end=RIGHT, color=BLUE, tip_shape=ArrowSquareTip)
    DR_group += Arrow(start=LEFT, end=RIGHT, color=BLUE, tip_shape=ArrowSquareFilledTip)
    DR_group += Arrow(start=LEFT, end=RIGHT, color=YELLOW, tip_shape=ArrowCircleTip)
    DR_group += Arrow(start=LEFT, end=RIGHT, color=YELLOW, tip_shape=ArrowCircleFilledTip)
    DR_group.arrange(DOWN)
    DR_group.move_to(4 * RIGHT + 2 * DOWN)

    self.add(left_group, middle_group, UR_group, DR_group)
"""

def __init__(
    self,
    *args: Any,
    stroke_width: float = 6,
    buff: float = MED_SMALL_BUFF,
    max_tip_length_to_length_ratio: float = 0.25,
    max_stroke_width_to_length_ratio: float = 5,
    **kwargs,
) :
    self.max_tip_length_to_length_ratio = max_tip_length_to_length_ratio
    self.max_stroke_width_to_length_ratio = max_stroke_width_to_length_ratio
    tip_shape = kwargs.pop("tip_shape", ArrowTriangleFilledTip)
    super().__init__(*args, buff=buff, stroke_width=stroke_width, **kwargs)
    # TODO, should this be affected when
    # Arrow.set_stroke is called?
    self.initial_stroke_width = self.stroke_width
    self.add_tip(tip_shape=tip_shape)
    self._set_stroke_width_from_length()

def scale(self, factor, scale_tips=False, **kwargs):
    """Scale an arrow, but keep stroke width and arrow tip size fixed.

    .. seealso::
        :meth:`~.Mobject.scale`

    Examples
    ======
    ::

        >>> arrow = Arrow(np.array([-1, -1, 0]), np.array([1, 1, 0]), buff=0)
        >>> scaled_arrow = arrow.scale(2)
        >>> np.round(scaled_arrow.get_start_and_end(), 8) + 0
        array([[-2., -2.,  0.],
               [ 2.,  2.,  0.]])
        >>> arrow.tip.length == scaled_arrow.tip.length
        True

    Manually scaling the object using the default method
    :meth:`~.Mobject.scale` does not have the same properties:

        >>> new_arrow = Arrow(np.array([-1, -1, 0]), np.array([1, 1, 0]), buff=0)
        >>> another_scaled_arrow = VMobject.scale(new_arrow, 2)
        >>> another_scaled_arrow.tip.length == arrow.tip.length
        False
    """

    if self.get_length() == 0:
        return self

    if scale_tips:
        super().scale(factor, **kwargs)
        self._set_stroke_width_from_length()
        return self

    has_tip = self.has_tip()
    has_start_tip = self.has_start_tip()
    if has_tip or has_start_tip:
        old_tips = self.pop_tips()

    super().scale(factor, **kwargs)
    self._set_stroke_width_from_length()

    if has_tip:
        self.add_tip(tip=old_tips[0])
    if has_start_tip:
        self.add_tip(tip=old_tips[1], at_start=True)
    return self

def get_normal_vector(self) -> np.ndarray:
    """Returns the normal of a vector.

    Examples
    ======
    ::

        >>> np.round(Arrow().get_normal_vector()) + 0. # add 0. to avoid negative 0 in output
        array([ 0.,  0., -1.])
    """

    p0, p1, p2 = self.tip.get_start_anchors()[:3]
    return normalize(np.cross(p2 - p1, p1 - p0))

def reset_normal_vector(self):
    """Resets the normal of a vector"""
    self.normal_vector = self.get_normal_vector()
    return self

def get_default_tip_length(self) -> float:
    """Returns the default tip_length of the arrow.

```

```

Examples
-----
:::

>>> Arrow().get_default_tip_length()
0.35
"""

max_ratio = self.max_tip_length_to_length_ratio
return min(self.tip_length, max_ratio * self.get_length())

def _set_stroke_width_from_length(self):
    """Sets stroke width based on length."""
    max_ratio = self.max_stroke_width_to_length_ratio
    if config.renderer == RendererType.OPENGGL:
        self.set_stroke(
            width=min(self.initial_stroke_width, max_ratio * self.get_length()),
            recurse=False,
        )
    else:
        self.set_stroke(
            width=min(self.initial_stroke_width, max_ratio * self.get_length()),
            family=False,
        )
    return self

class Vector(Arrow):
    """A vector specialized for use in graphs.

Parameters
-----
direction
    The direction of the arrow.
buff
    The distance of the vector from its endpoints.
kwargs
    Additional arguments to be passed to :class:`Arrow`


Examples
-----
.. manim:: VectorExample
    :save_last_frame:

    class VectorExample(Scene):
        def construct(self):
            plane = NumberPlane()
            vector_1 = Vector([1,2])
            vector_2 = Vector([-5,-2])
            self.add(plane, vector_1, vector_2)

    """

def __init__(self, direction: list | np.ndarray = RIGHT, buff: float = 0, **kwargs):
    self.buff = buff
    if len(direction) == 2:
        direction = np.hstack([direction, 0])

    super().__init__(ORIGIN, direction, buff=buff, **kwargs)

def coordinate_label(
    self,
    integer_labels: bool = True,
    n_dim: int = 2,
    color: Color | None = None,
    **kwargs,
):
    """Creates a label based on the coordinates of the vector.

Parameters
-----
integer_labels
    Whether or not to round the coordinates to integers.
n_dim
    The number of dimensions of the vector.
color
    Sets the color of label, optional.
kwargs
    Additional arguments to be passed to :class:`~.Matrix`.


Returns
-----
:class:`~.Matrix`
    The label.

Examples
-----
.. manim:: VectorCoordinateLabel
    :save_last_frame:

    class VectorCoordinateLabel(Scene):
        def construct(self):
            plane = NumberPlane()

            vec_1 = Vector([1, 2])
            vec_2 = Vector([-3, -2])
            label_1 = vec_1.coordinate_label()
            label_2 = vec_2.coordinate_label(color=WHITE)

            self.add(plane, vec_1, vec_2, label_1, label_2)

    """

# avoiding circular imports
from ..matrix import Matrix

vect = np.array(self.get_end())
if integer_labels:
    vect = np.round(vect).astype(int)
vect = vect[:n_dim]
vect = vect.reshape((n_dim, 1))
label = Matrix(vect, **kwargs)
label.scale(LARGE_BUFF - 0.2)

shift_dir = np.array(self.get_end())
if shift_dir[0] >= 0: # Pointing right
    shift_dir -= label.get_left() + DEFAULT_MOBJECT_TO_MOBJECT_BUFFER * LEFT
else: # Pointing left

```

```

    shift_dir -= label.get_right() + DEFAULT_MOBJECT_TO_MOBJECT_BUFFER * RIGHT
label.shift(shift_dir)
if color is not None:
    label.set_color(color)
return label

class DoubleArrow(Arrow):
    """An arrow with tips on both ends.

Parameters
-----
args
    Arguments to be passed to :class:`Arrow`
kwargs
    Additional arguments to be passed to :class:`Arrow`

.. seealso::
    :class:`~ArrowTip`
    :class:`~CurvedDoubleArrow`

Examples
-----
.. manim:: DoubleArrowExample
    :save_last_frame:

from manim.mobject.geometry.tips import ArrowCircleFilledTip
class DoubleArrowExample(Scene):
    def construct(self):
        circle = Circle(radius=2.0)
        d_arrow = DoubleArrow(start=circle.get_left(), end=circle.get_right())
        d_arrow_2 = DoubleArrow(tip_shape_end=ArrowCircleFilledTip, tip_shape_start=ArrowCircleFilledTip)
        group = Group(Group(circle, d_arrow), d_arrow_2).arrange(UP, buff=1)
        self.add(group)

.. manim:: DoubleArrowExample2
    :save_last_frame:

class DoubleArrowExample2(Scene):
    def construct(self):
        box = Square()
        p1 = box.get_left()
        p2 = box.get_right()
        d1 = DoubleArrow(p1, p2, buff=0)
        d2 = DoubleArrow(p1, p2, buff=0, tip_length=0.2, color=YELLOW)
        d3 = DoubleArrow(p1, p2, buff=0, tip_length=0.4, color=BLUE)
        Group(d1, d2, d3).arrange(DOWN)
        self.add(box, d1, d2, d3)
"""

def __init__(self, *args: Any, **kwargs):
    if "tip_shape_end" in kwargs:
        kwargs["tip_shape"] = kwargs.pop("tip_shape_end")
    tip_shape_start = kwargs.pop("tip_shape_start", ArrowTriangleFilledTip)
    super().__init__(*args, **kwargs)
    self.add_tip(at_start=True, tip_shape=tip_shape_start)

class Angle(VMobject, metaclass=ConvertToOpenGL):
    """A circular arc or elbow-type mobobject representing an angle of two lines.

Parameters
-----
line1 :
    The first line.
line2 :
    The second line.
radius :
    The radius of the :class:`Arc`.
quadrant
    A sequence of two :class:`int` numbers determining which of the 4 quadrants should be used.
    The first value indicates whether to anchor the arc on the first line closer to the end point (1)
    or start point (-1), and the second value functions similarly for the
    end (1) or start (-1) of the second line.
    Possibilities: (1,1), (-1,1), (1,-1), (-1,-1).
other_angle :
    Toggles between the two possible angles defined by two points and an arc center. If set to
    False (default), the arc will always go counterclockwise from the point on line1 until
    the point on line2 is reached. If set to True, the angle will go clockwise from line1 to line2.
dot
    Allows for a :class:`Dot` in the arc. Mainly used as an convention to indicate a right angle.
    The dot can be customized in the next three parameters.
dot_radius
    The radius of the :class:`Dot`. If not specified otherwise, this radius will be 1/10 of the arc radius.
dot_distance
    Relative distance from the center to the arc: 0 puts the dot in the center and 1 on the arc itself.
dot_color
    The color of the :class:`Dot`.
elbow
    Produces an elbow-type mobobject indicating a right angle, see :class:`RightAngle` for more information
    and a shorthand.
**kwargs
    Further keyword arguments that are passed to the constructor of :class:`Arc` or :class:`Elbow`.

Examples
-----
The first example shows some right angles with a dot in the middle while the second example shows
all 8 possible angles defined by two lines.

.. manim:: RightArcAngleExample
    :save_last_frame:

class RightArcAngleExample(Scene):
    def construct(self):
        line1 = Line(LEFT, RIGHT)
        line2 = Line(DOWN, UP)
        rightarcangles = [
            Angle(line1, line2, dot=True),
            Angle(line1, line2, radius=0.4, quadrant=(1,-1), dot=True, other_angle=True),
            Angle(line1, line2, radius=0.5, quadrant=(-1,1), stroke_width=8, dot=True, dot_color=YELLOW, dot_radius=0.04, other_angle=True),
            Angle(line1, line2, radius=0.7, quadrant=(-1,-1), color=RED, dot=True, dot_color=GREEN, dot_radius=0.08),
        ]
        plots = VGroup()
        for angle in rightarcangles:
            plot=VGroup(line1.copy(), line2.copy(), angle)
            plots.add(plot)

```

```

plots.arrange(buff=1.5)
self.add(plots)

.. manim:: AngleExample
:save_last_frame:

class AngleExample(Scene):
    def construct(self):
        line1 = Line( LEFT + (1/3) * UP, RIGHT + (1/3) * DOWN )
        line2 = Line( DOWN + (1/3) * RIGHT, UP + (1/3) * LEFT )
        angles = [
            Angle(line1, line2),
            Angle(line1, line2, radius=0.4, quadrant=(1,-1), other_angle=True),
            Angle(line1, line2, radius=0.5, quadrant=(-1,1), stroke_width=8, other_angle=True),
            Angle(line1, line2, radius=0.7, quadrant=(-1,-1), color=RED),
            Angle(line1, line2, other_angle=True),
            Angle(line1, line2, radius=0.4, quadrant=(1,-1)),
            Angle(line1, line2, radius=0.5, quadrant=(-1,1), stroke_width=8),
            Angle(line1, line2, radius=0.7, quadrant=(-1,-1), color=RED, other_angle=True),
        ]
        plots = VGroup()
        for angle in angles:
            plot=VGroup(line1.copy(),line2.copy(), angle)
            plots.add(VGroup(plot, SurroundingRectangle(plot, buff=0.3)))
        plots.arrange_in_grid(rows=2,buff=1)
        self.add(plots)

.. manim:: FilledAngle
:save_last_frame:

class FilledAngle(Scene):
    def construct(self):
        l1 = Line(ORIGIN, 2 * UP + RIGHT).set_color(GREEN)
        l2 = (
            Line(ORIGIN, 2 * UP + RIGHT)
            .set_color(GREEN)
            .rotate(-20 * DEGREES, about_point=ORIGIN)
        )
        norm = l1.get_length()
        a1 = Angle(l1, l2, other_angle=True, radius=norm - 0.5).set_color(GREEN)
        a2 = Angle(l1, l2, other_angle=True, radius=norm).set_color(GREEN)
        q1 = a1.points # save all coordinates of points of angle a1
        q2 = a2.reverse_direction().points # save all coordinates of points of angle a1 (in reversed direction)
        pnts = np.concatenate([q1, q2, q1[0].reshape(1, 3)]) # adds points and ensures that path starts and ends at same point
        mfill = VMobject().set_color(ORANGE)
        mfill.set_points_as_corners(pnts).set_fill(GREEN, opacity=1)
        self.add(l1, l2)
        self.add(mfill)

"""

def __init__(
    self,
    line1: Line,
    line2: Line,
    radius: float = None,
    quadrant: Sequence[int] = (1, 1),
    other_angle: bool = False,
    dot: bool = False,
    dot_radius: float | None = None,
    dot_distance: float = 0.55,
    dot_color: Colors = WHITE,
    elbow: bool = False,
    **kwargs,
):
    super().__init__(**kwargs)
    self.lines = (line1, line2)
    self.quadrant = quadrant
    self.dot_distance = dot_distance
    self.elbow = elbow
    inter = line_intersection(
        [line1.get_start(), line1.get_end()],
        [line2.get_start(), line2.get_end()],
    )

    if radius is None:
        if quadrant[0] == 1:
            dist_1 = np.linalg.norm(line1.get_end() - inter)
        else:
            dist_1 = np.linalg.norm(line1.get_start() - inter)
        if quadrant[1] == 1:
            dist_2 = np.linalg.norm(line2.get_end() - inter)
        else:
            dist_2 = np.linalg.norm(line2.get_start() - inter)
        if np.minimum(dist_1, dist_2) < 0.6:
            radius = (2 / 3) * np.minimum(dist_1, dist_2)
        else:
            radius = 0.4
    else:
        self.radius = radius

    anchor_angle_1 = inter + quadrant[0] * radius * line1.get_unit_vector()
    anchor_angle_2 = inter + quadrant[1] * radius * line2.get_unit_vector()

    if elbow:
        anchor_middle = (
            inter
            + quadrant[0] * radius * line1.get_unit_vector()
            + quadrant[1] * radius * line2.get_unit_vector()
        )
        angle_mobject = Elbow(**kwargs)
        angle_mobject.set_points_as_corners(
            [anchor_angle_1, anchor_middle, anchor_angle_2],
        )
    else:
        angle_1 = angle_of_vector(anchor_angle_1 - inter)
        angle_2 = angle_of_vector(anchor_angle_2 - inter)

        if not other_angle:
            start_angle = angle_1
            if angle_2 > angle_1:
                angle_fin = angle_2 - angle_1
            else:
                angle_fin = 2 * np.pi - (angle_1 - angle_2)
        else:
            start_angle = angle_1
            if angle_2 < angle_1:
                angle_fin = -angle_1 + angle_2

```

```

        else:
            angle_fin = -2 * np.pi + (angle_2 - angle_1)
        self.angle_value = angle_fin

        angle_mobject = Arc(
            radius=radius,
            angle=self.angle_value,
            start_angle=start_angle,
            arc_center=inter,
            **kwargs,
        )

        if dot:
            if dot_radius is None:
                dot_radius = radius / 10
            else:
                self.dot_radius = dot_radius
            right_dot = Dot(ORIGIN, radius=dot_radius, color=dot_color)
            dot_anchor = (
                inter
                + (angle_mobject.get_center() - inter)
                / np.linalg.norm(angle_mobject.get_center() - inter)
                * radius
                * dot_distance
            )
            right_dot.move_to(dot_anchor)
            self.add(right_dot)

        self.set_points(angle_mobject.points)

    def get_lines(self) -> VGroup:
        """Get the lines forming an angle of the :class:`Angle` class.

        Returns
        -------
        :class:`~.VGroup`
            A :class:`~.VGroup` containing the lines that form the angle of the :class:`Angle` class.

        Examples
        -----
        ::

            >>> line_1, line_2 = Line(ORIGIN, RIGHT), Line(ORIGIN, UR)
            >>> angle = Angle(line_1, line_2)
            >>> angle.get_lines()
            VGroup(Line, Line)
        """

        return VGroup(*self.lines)

    def get_value(self, degrees: bool = False) -> float:
        """Get the value of an angle of the :class:`Angle` class.

        Parameters
        -----
        degrees
            A boolean to decide the unit (deg/rad) in which the value of the angle is returned.

        Returns
        -------
        :class:`float`
            The value in degrees/radians of an angle of the :class:`Angle` class.

        Examples
        -----
        .. manim:: GetValueExample
           :save_last_frame:

            class GetValueExample(Scene):
                def construct(self):
                    line1 = Line(LEFT+(1/3)*UP, RIGHT+(1/3)*DOWN)
                    line2 = Line(DOWN+(1/3)*RIGHT, UP+(1/3)*LEFT)

                    angle = Angle(line1, line2, radius=0.4)

                    value = DecimalNumber(angle.get_value(degrees=True), unit="^{\circ}")
                    value.next_to(angle, UR)

                    self.add(line1, line2, angle, value)
        """

        if degrees:
            return self.angle_value / DEGREES
        return self.angle_value

    @staticmethod
    def from_three_points(
        A: np.ndarray, B: np.ndarray, C: np.ndarray, **kwargs
    ) -> Angle:
        """The angle between the lines AB and BC.

        This constructs the angle :math:`\angle ABC`.

        Parameters
        -----
        A
            The endpoint of the first angle leg
        B
            The vertex of the angle
        C
            The endpoint of the second angle leg
        **kwargs
            Further keyword arguments are passed to :class:`Angle`

        Returns
        -----
        The Angle calculated from the three points
        Angle(line1, line2, radius=0.5, quadrant=(-1,1), stroke_width=8),
        Angle(line1, line2, radius=0.7, quadrant=(-1,-1), color=RED, other_angle=True),
        """

        Examples
        -----
        .. manim:: AngleFromThreePointsExample
           :save_last_frame:

```

```

class AngleFromThreePointsExample(Scene):
    def construct(self):
        sample_angle = Angle.from_three_points(UP, ORIGIN, LEFT)
        red_angle = Angle.from_three_points(LEFT + UP, ORIGIN, RIGHT, radius=.8, quadrant=(-1,-1), color=RED, stroke_width=8, other_angle=True)
        self.add(red_angle, sample_angle)
    """
    return Angle(Line(B, A), Line(B, C), **kwargs)

class RightAngle(Angle):
    """An elbow-type mobject representing a right angle between two lines.

    Parameters
    -----
    line1
        The first line.
    line2
        The second line.
    length
        The length of the arms.
    **kwargs
        Further keyword arguments that are passed to the constructor of :class:`Angle`.

    Examples
    -----
    .. manim:: RightAngleExample
        :save_last_frame:

    class RightAngleExample(Scene):
        def construct(self):
            line1 = Line( LEFT, RIGHT )
            line2 = Line( DOWN, UP )
            rightangles = [
                RightAngle(line1, line2),
                RightAngle(line1, line2, length=0.4, quadrant=(1,-1)),
                RightAngle(line1, line2, length=0.5, quadrant=(-1,1), stroke_width=8),
                RightAngle(line1, line2, length=0.7, quadrant=(-1,-1), color=RED),
            ]
            plots = VGroup()
            for rightangle in rightangles:
                plot=VGroup(line1.copy(),line2.copy(), rightangle)
                plots.add(plot)
            plots.arrange(buff=1.5)
            self.add(plots)
    """
    def __init__(self, line1: Line, line2: Line, length: float | None = None, **kwargs):
        super().__init__(line1, line2, radius=length, elbow=True, **kwargs)

```

show_diff.py

```
from __future__ import annotations

import logging
import warnings

import numpy as np

def show_diff_helper(
    frame_number: int,
    frame_data: np.ndarray,
    expected_frame_data: np.ndarray,
    control_data_filename: str,
):
    """Will visually display with matplotlib differences between frame generated and the one expected."""
    import matplotlib.gridspec as gridspec
    import matplotlib.pyplot as plt

    gs = gridspec.GridSpec(2, 2)
    fig = plt.figure()
    fig.suptitle(f"Test difference summary at frame {frame_number}", fontsize=16)

    ax = fig.add_subplot(gs[0, 0])
    ax.imshow(frame_data)
    ax.set_title("Generated")

    ax = fig.add_subplot(gs[0, 1])
    ax.imshow(expected_frame_data)
    ax.set_title("Expected")

    ax = fig.add_subplot(gs[1, :])
    diff_im = expected_frame_data.copy()
    diff_im = np.where(
        frame_data != np.array([0, 0, 0, 255]),
        np.array([0, 255, 0, 255], dtype="uint8"),
        np.array([0, 0, 0, 255], dtype="uint8"),
    ) # Set any non-black pixels to green
    np.putmask(
        diff_im,
        expected_frame_data != frame_data,
        np.array([255, 0, 0, 255], dtype="uint8"),
    ) # Set any different pixels to red
    ax.imshow(diff_im, interpolation="nearest")
    ax.set_title("Difference summary: (green = same, red = different)")

    with warnings.catch_warnings():
        warnings.simplefilter("error")
        try:
            plt.show()
        except UserWarning:
            filename = f"{control_data_filename[:-4]}-diff.pdf"
            plt.savefig(filename)
            logging.warning(
                "Interactive matplotlib interface not available,"
                f" diff saved to {filename}."
            )
    
```

config_ops.py

```
"""Utilities that might be useful for configuration dictionaries."""

from __future__ import annotations

__all__ = [
    "merge_dicts_recursively",
    "update_dict_recursively",
    "DictAsObject",
]
]

import itertools as it
import numpy as np

def merge_dicts_recursively(*dicts):
    """
    Creates a dict whose keyset is the union of all the input dictionaries. The value for each key is based on the first dict in the list with that key.

    dicts later in the list have higher priority
    When values are dictionaries, it is applied recursively
    """
    result = {}
    all_items = it.chain(*(d.items() for d in dicts))
    for key, value in all_items:
        if key in result and isinstance(result[key], dict) and isinstance(value, dict):
            result[key] = merge_dicts_recursively(result[key], value)
        else:
            result[key] = value
    return result

def update_dict_recursively(current_dict, *others):
    updated_dict = merge_dicts_recursively(current_dict, *others)
    current_dict.update(updated_dict)

# Occasionally convenient in order to write dict.x instead of more laborious
# (and less in keeping with all other attr accesses) dict["x"]

class DictAsObject:
    def __init__(self, dictin):
        self.__dict__ = dictin

class _Data:
    """
    Descriptor that allows _Data variables to be grouped and accessed from self.data["attr"] via self.attr. self.data attributes must be arrays.
    """
    def __set_name__(self, obj, name):
        self.name = name

    def __get__(self, obj, owner):
        return obj.data[self.name]

    def __set__(self, obj, array: np.ndarray):
        obj.data[self.name] = array

class _Uniforms:
    """
    Descriptor that allows _Uniforms variables to be grouped from self.uniforms["attr"] via self.attr. self.uniforms attributes must be floats.
    """
    def __set_name__(self, obj, name):
        self.name = name

    def __get__(self, obj, owner):
        return obj.__dict__["uniforms"][self.name]

    def __set__(self, obj, num: float):
        obj.__dict__["uniforms"][self.name] = num
```

arc.py

```
r"""Mobjects that are curved.

Examples
-----
.. manim:: UsefulAnnotations
    :save_last_frame:

    class UsefulAnnotations(Scene):
        def construct(self):
            m0 = Dot()
            m1 = AnnotationDot()
            m2 = LabeledDot("ii")
            m3 = LabeledDot(MathTex(r"\alpha").set_color(ORANGE))
            m4 = CurvedArrow(2*LEFT, 2*RIGHT, radius= -5)
            m5 = CurvedArrow(2*LEFT, 2*RIGHT, radius= 8)
            m6 = CurvedDoubleArrow(ORIGIN, 2*RIGHT)

            self.add(m0, m1, m2, m3, m4, m5, m6)
            for i, mobj in enumerate(self.mobjects):
                mobj.shift(DOWN * (i-3))

"""

from __future__ import annotations

__all__ = [
    "TipableVMobject",
    "Arc",
    "ArcBetweenPoints",
    "CurvedArrow",
    "CurvedDoubleArrow",
    "Circle",
    "Dot",
    "AnnotationDot",
    "LabeledDot",
    "Ellipse",
    "AnnularSector",
    "Sector",
    "Annulus",
    "CubicBezier",
    "ArcPolygon",
    "ArcPolygonFromArcs",
]
import itertools
import math
import warnings
from typing import TYPE_CHECKING, Sequence

import numpy as np
from colour import Color

from manim.constants import *
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.types.vectorized_mobject import VMobject
from manim.utils.color import *
from manim.utils.iterables import adjacent_pairs
from manim.utils.space_ops import (
    angle_of_vector,
    cartesian_to_spherical,
    line_intersection,
    perpendicular_bisector,
    rotate_vector,
)
if TYPE_CHECKING:
    from manim.mobject.mobject import Mobject
    from manim.mobject.text.tex_mobject import SingleStringMathTex, Tex
    from manim.mobject.text.text_mobject import Text

class TipableVMobject(VMobject, metaclass=ConvertToOpenGL):
    """Meant for shared functionality between Arc and Line.
    Functionality can be classified broadly into these groups:

    * Adding, Creating, Modifying tips
        - add_tip calls create_tip, before pushing the new tip
            into the TipableVMobject's list of submobjects
        - stylistic and positional configuration

    * Checking for tips
        - Boolean checks for whether the TipableVMobject has a tip
            and a starting tip

    * Getters
        - Straightforward accessors, returning information pertaining
            to the TipableVMobject instance's tip(s), its length etc
"""

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self._tips = []
        self._tip_index = 0
        self._tip_start = None
        self._tip_end = None
        self._tip_length = 0.0
        self._tip_color = Color("black")
```

```

def __init__(
    self,
    tip_length=DEFAULT_ARROW_TIP_LENGTH,
    normal_vector=OUT,
    tip_style={},
    **kwargs,
):
    self.tip_length = tip_length
    self.normal_vector = normal_vector
    self.tip_style = tip_style
    super().__init__(**kwargs)

# Adding, Creating, Modifying tips

def add_tip(
    self, tip=None, tip_shape=None, tip_length=None, tip_width=None, at_start=False
):
    """Adds a tip to the TipableVMobject instance, recognising
    that the endpoints might need to be switched if it's
    a 'starting tip' or not.
    """
    if tip is None:
        tip = self.create_tip(tip_shape, tip_length, tip_width, at_start)
    else:
        self.position_tip(tip, at_start)
    self.reset_endpoints_based_on_tip(tip, at_start)
    self.assign_tip_attr(tip, at_start)
    self.add(tip)
    return self

def create_tip(
    self, tip_shape=None, tip_length=None, tip_width=None, at_start=False
):
    """Stylises the tip, positions it spatially, and returns
    the newly instantiated tip to the caller.
    """
    tip = self.get_unpositioned_tip(tip_shape, tip_length, tip_width)
    self.position_tip(tip, at_start)
    return tip

def get_unpositioned_tip(self, tip_shape=None, tip_length=None, tip_width=None):
    """Returns a tip that has been stylistically configured,
    but has not yet been given a position in space.
    """
    from manim.mobject.geometry.tips import ArrowTriangleFilledTip

    style = {}

    if tip_shape is None:
        tip_shape = ArrowTriangleFilledTip

    if tip_shape is ArrowTriangleFilledTip:
        if tip_width is None:
            tip_width = self.get_default_tip_length()
        style.update({"width": tip_width})

    if tip_length is None:
        tip_length = self.get_default_tip_length()

    color = self.get_color()
    style.update({"fill_color": color, "stroke_color": color})
    style.update(self.tip_style)
    tip = tip_shape(length=tip_length, **style)
    return tip

def position_tip(self, tip, at_start=False):
    # Last two control points, defining both
    # the end, and the tangency direction
    if at_start:
        anchor = self.get_start()
        handle = self.get_first_handle()
    else:
        handle = self.get_last_handle()
        anchor = self.get_end()
    angles = cartesian_to_spherical(handle - anchor)
    tip.rotate(
        angles[1] - PI - tip.tip_angle,
    ) # Rotates the tip along the azimuthal
    if not hasattr(self, "_init_positioning_axis"):
        axis = [
            np.sin(angles[1]),
            -np.cos(angles[1]),
            0,
        ] # Obtains the perpendicular of the tip
        tip.rotate(
            -angles[2] + PI / 2,
            axis=axis,
        ) # Rotates the tip along the vertical wrt the axis
        self._init_positioning_axis = axis
    tip.shift(anchor - tip.tip_point)
    return tip

def reset_endpoints_based_on_tip(self, tip, at_start):
    if self.get_length() == 0:
        # Zero length, put_start_and_end_on wouldn't work
        return self

```

```

    if at_start:
        self.put_start_and_end_on(tip.base, self.get_end())
    else:
        self.put_start_and_end_on(self.get_start(), tip.base)
    return self

def asign_tip_attr(self, tip, at_start):
    if at_start:
        self.start_tip = tip
    else:
        self.tip = tip
    return self

# Checking for tips

def has_tip(self):
    return hasattr(self, "tip") and self.tip in self

def has_start_tip(self):
    return hasattr(self, "start_tip") and self.start_tip in self

# Getters

def pop_tips(self):
    start, end = self.get_start_and_end()
    result = self.get_group_class()()
    if self.has_tip():
        result.add(self.tip)
        self.remove(self.tip)
    if self.has_start_tip():
        result.add(self.start_tip)
        self.remove(self.start_tip)
    self.put_start_and_end_on(start, end)
    return result

def get_tips(self):
    """Returns a VGroup (collection of VMobjects) containing
    the TipableVMObject instance's tips.
    """
    result = self.get_group_class()()
    if hasattr(self, "tip"):
        result.add(self.tip)
    if hasattr(self, "start_tip"):
        result.add(self.start_tip)
    return result

def get_tip(self):
    """Returns the TipableVMobject instance's (first) tip,
    otherwise throws an exception."""
    tips = self.get_tips()
    if len(tips) == 0:
        raise Exception("tip not found")
    else:
        return tips[0]

def get_default_tip_length(self):
    return self.tip_length

def get_first_handle(self):
    return self.points[1]

def get_last_handle(self):
    return self.points[-2]

def get_end(self):
    if self.has_tip():
        return self.tip.get_start()
    else:
        return super().get_end()

def get_start(self):
    if self.has_start_tip():
        return self.start_tip.get_start()
    else:
        return super().get_start()

def get_length(self):
    start, end = self.get_start_and_end()
    return np.linalg.norm(start - end)

class Arc(TipableVMobject):
    """A circular arc.

Examples
-----
A simple arc of angle Pi.

.. manim:: ArcExample
    :save_last_frame:

    class ArcExample(Scene):
        def construct(self):
            self.add(Arc(angle=PI))

```

```

"""
def __init__(self,
            radius: float = 1.0,
            start_angle=0,
            angle=TAU / 4,
            num_components=9,
            arc_center=ORIGIN,
            **kwargs,
            ):
    if radius is None: # apparently None is passed by ArcBetweenPoints
        radius = 1.0
    self.radius = radius
    self.num_components = num_components
    self.arc_center = arc_center
    self.start_angle = start_angle
    self.angle = angle
    self._failed_to_get_center = False
    super().__init__(**kwargs)

def generate_points(self):
    self._set_pre_positioned_points()
    self.scale(self.radius, about_point=ORIGIN)
    self.shift(self.arc_center)

# Points are set a bit differently when rendering via OpenGL.
# TODO: refactor Arc so that only one strategy for setting points
# has to be used.
def init_points(self):
    self.set_points(
        Arc._create_quadratic_bezier_points(
            angle=self.angle,
            start_angle=self.start_angle,
            n_components=self.num_components,
        ),
    )
    self.scale(self.radius, about_point=ORIGIN)
    self.shift(self.arc_center)

@staticmethod
def _create_quadratic_bezier_points(angle, start_angle=0, n_components=8):
    samples = np.array([
        [np.cos(a), np.sin(a), 0]
        for a in np.linspace(
            start_angle,
            start_angle + angle,
            2 * n_components + 1,
        )
    ],
    )
    theta = angle / n_components
    samples[1::2] /= np.cos(theta / 2)

    points = np.zeros((3 * n_components, 3))
    points[0::3] = samples[0:-1:2]
    points[1::3] = samples[1::2]
    points[2::3] = samples[2::2]
    return points

def _set_pre_positioned_points(self):
    anchors = np.array([
        [
            np.cos(a) * RIGHT + np.sin(a) * UP
            for a in np.linspace(
                self.start_angle,
                self.start_angle + self.angle,
                self.num_components,
            )
        ],
    ],
    )
    # Figure out which control points will give the
    # appropriate tangent lines to the circle
    d_theta = self.angle / (self.num_components - 1.0)
    tangent_vectors = np.zeros(anchors.shape)
    # Rotate all 90 degrees, via (x, y) -> (-y, x)
    tangent_vectors[:, 1] = anchors[:, 0]
    tangent_vectors[:, 0] = -anchors[:, 1]
    # Use tangent vectors to deduce anchors
    handles1 = anchors[:-1] + (d_theta / 3) * tangent_vectors[:-1]
    handles2 = anchors[1:] - (d_theta / 3) * tangent_vectors[1:]
    self.set_anchors_and_handles(anchors[:-1], handles1, handles2, anchors[1:])

def get_arc_center(self, warning=True):
    """
    Looks at the normals to the first two
    anchors, and finds their intersection points
    """
    # First two anchors and handles
    a1, h1, h2, a2 = self.points[:4]

    if np.all(a1 == a2):
        # For a1 and a2 to lie at the same point arc radius
        # must be zero. Thus arc_center will also lie at
        # that point.

```

```

        return a1
    # Tangent vectors
    t1 = h1 - a1
    t2 = h2 - a2
    # Normals
    n1 = rotate_vector(t1, TAU / 4)
    n2 = rotate_vector(t2, TAU / 4)
    try:
        return line_intersection(line1=(a1, a1 + n1), line2=(a2, a2 + n2))
    except Exception:
        if warning:
            warnings.warn("Can't find Arc center, using ORIGIN instead")
        self._failed_to_get_center = True
        return np.array(ORIGIN)

def move_arc_center_to(self, point):
    self.shift(point - self.get_arc_center())
    return self

def stop_angle(self):
    return angle_of_vector(self.points[-1] - self.get_arc_center()) % TAU

class ArcBetweenPoints(Arc):
    """Inherits from Arc and additionally takes 2 points between which the arc is spanned.

Example
-----
.. manim:: ArcBetweenPointsExample

class ArcBetweenPointsExample(Scene):
    def construct(self):
        circle = Circle(radius=2, stroke_color=GREY)
        dot_1 = Dot(color=GREEN).move_to([2, 0, 0]).scale(0.5)
        dot_1_text = Tex("(2,0)").scale(0.5).next_to(dot_1, RIGHT).set_color(BLUE)
        dot_2 = Dot(color=GREEN).move_to([0, 2, 0]).scale(0.5)
        dot_2_text = Tex("(0,2)").scale(0.5).next_to(dot_2, UP).set_color(BLUE)
        arc= ArcBetweenPoints(start=2 * RIGHT, end=2 * UP, stroke_color=YELLOW)
        self.add(circle, dot_1, dot_2, dot_1_text, dot_2_text)
        self.play(Create(arc))
"""

def __init__(self, start, end, angle=TAU / 4, radius=None, **kwargs):
    if radius is not None:
        self.radius = radius
        if radius < 0:
            sign = -2
            radius *= -1
    else:
        sign = 2
    halfdist = np.linalg.norm(np.array(start) - np.array(end)) / 2
    if radius < halfdist:
        raise ValueError(
            """ArcBetweenPoints called with a radius that is
            smaller than half the distance between the points."""
        )
    arc_height = radius - math.sqrt(radius**2 - halfdist**2)
    angle = math.acos((radius - arc_height) / radius) * sign

    super().__init__(radius=radius, angle=angle, **kwargs)
    if angle == 0:
        self.set_points_as_corners([LEFT, RIGHT])
        self.put_start_and_end_on(start, end)

    if radius is None:
        center = self.get_arc_center(warning=False)
        if not self._failed_to_get_center:
            self.radius = np.linalg.norm(np.array(start) - np.array(center))
        else:
            self.radius = math.inf

class CurvedArrow(ArcBetweenPoints):
    def __init__(self, start_point, end_point, **kwargs):
        from manim.mobject.geometry.tips import ArrowTriangleFilledTip

        tip_shape = kwargs.pop("tip_shape", ArrowTriangleFilledTip)
        super().__init__(start_point, end_point, **kwargs)
        self.add_tip(tip_shape=tip_shape)

class CurvedDoubleArrow(CurvedArrow):
    def __init__(self, start_point, end_point, **kwargs):
        if "tip_shape_end" in kwargs:
            kwargs["tip_shape"] = kwargs.pop("tip_shape_end")
        from manim.mobject.geometry.tips import ArrowTriangleFilledTip

        tip_shape_start = kwargs.pop("tip_shape_start", ArrowTriangleFilledTip)
        super().__init__(start_point, end_point, **kwargs)
        self.add_tip(at_start=True, tip_shape=tip_shape_start)

class Circle(Arc):
    """A circle.

```

```

Parameters
-----
color
    The color of the shape.
kwargs
    Additional arguments to be passed to :class:`Arc`


Examples
-----
.. manim:: CircleExample
    :save_last_frame:

    class CircleExample(Scene):
        def construct(self):
            circle_1 = Circle(radius=1.0)
            circle_2 = Circle(radius=1.5, color=GREEN)
            circle_3 = Circle(radius=1.0, color=BLUE_B, fill_opacity=1)

            circle_group = Group(circle_1, circle_2, circle_3).arrange(buff=1)
            self.add(circle_group)

    """


def __init__(
    self,
    radius: float | None = None,
    color: Color | str = RED,
    **kwargs,
):
    super().__init__(
        radius=radius,
        start_angle=0,
        angle=TAU,
        color=color,
        **kwargs,
    )

def surround(
    self,
    mobject: Mobject,
    dim_to_match: int = 0,
    stretch: bool = False,
    buffer_factor: float = 1.2,
):
    """Modifies a circle so that it surrounds a given mobject.

Parameters
-----
mobject
    The mobject that the circle will be surrounding.
dim_to_match
buffer_factor
    Scales the circle with respect to the mobject. A `buffer_factor` < 1 makes the circle smaller than the mobject.
stretch
    Stretches the circle to fit more tightly around the mobject. Note: Does not work with :class:`Line`


Examples
-----
.. manim:: CircleSurround
    :save_last_frame:

    class CircleSurround(Scene):
        def construct(self):
            triangle1 = Triangle()
            circle1 = Circle().surround(triangle1)
            group1 = Group(triangle1, circle1) # treat the two mobjects as one

            line2 = Line()
            circle2 = Circle().surround(line2, buffer_factor=2.0)
            group2 = Group(line2, circle2)

            # buffer_factor < 1, so the circle is smaller than the square
            square3 = Square()
            circle3 = Circle().surround(square3, buffer_factor=0.5)
            group3 = Group(square3, circle3)

            group = Group(group1, group2, group3).arrange(buff=1)
            self.add(group)

    """
    # Ignores dim_to_match and stretch; result will always be a circle
    # TODO: Perhaps create an ellipse class to handle single-dimension stretching

    # Something goes wrong here when surrounding lines?
    # TODO: Figure out and fix
    self.replace(mobject, dim_to_match, stretch)

    self.width = np.sqrt(mobject.width**2 + mobject.height**2)
    return self.scale(buffer_factor)

def point_at_angle(self, angle: float):
    """Returns the position of a point on the circle.

Parameters
-----
angle

```

```

The angle of the point along the circle in radians.

Returns
-----
:class:`numpy.ndarray`
    The location of the point along the circle's circumference.

Examples
-----
.. manim:: PointAtAngleExample
    :save_last_frame:

    class PointAtAngleExample(Scene):
        def construct(self):
            circle = Circle(radius=2.0)
            p1 = circle.point_at_angle(PI/2)
            p2 = circle.point_at_angle(270*DEGREES)

            s1 = Square(side_length=0.25).move_to(p1)
            s2 = Square(side_length=0.25).move_to(p2)
            self.add(circle, s1, s2)

    """
    start_angle = angle_of_vector(self.points[0] - self.get_center())
    proportion = (angle - start_angle) / TAU
    proportion -= math.floor(proportion)
    return self.point_from_proportion(proportion)

@staticmethod
def from_three_points(
    p1: Sequence[float], p2: Sequence[float], p3: Sequence[float], **kwargs
):
    """Returns a circle passing through the specified three points.

Example
-----
.. manim:: CircleFromPointsExample
    :save_last_frame:

    class CircleFromPointsExample(Scene):
        def construct(self):
            circle = Circle.from_three_points(LEFT, LEFT + UP, UP * 2, color=RED)
            dots = VGroup(
                Dot(LEFT),
                Dot(LEFT + UP),
                Dot(UP * 2),
            )
            self.add(NumberPlane(), circle, dots)

    """
    center = line_intersection(
        perpendicular_bisector([p1, p2]),
        perpendicular_bisector([p2, p3]),
    )
    radius = np.linalg.norm(p1 - center)
    return Circle(radius=radius, **kwargs).shift(center)

class Dot(Circle):
    """A circle with a very small radius.

Parameters
-----
point
    The location of the dot.
radius
    The radius of the dot.
stroke_width
    The thickness of the outline of the dot.
fill_opacity
    The opacity of the dot's fill_colour
color
    The color of the dot.
kwargs
    Additional arguments to be passed to :class:`Circle`'s constructor.

Examples
-----
.. manim:: DotExample
    :save_last_frame:

    class DotExample(Scene):
        def construct(self):
            dot1 = Dot(point=LEFT, radius=0.08)
            dot2 = Dot(point=ORIGIN)
            dot3 = Dot(point=RIGHT)
            self.add(dot1, dot2, dot3)

    """
    def __init__(
        self,
        point: list | np.ndarray = ORIGIN,
        radius: float = DEFAULT_DOT_RADIUS,
        stroke_width: float = 0,

```

```

fill_opacity: float = 1.0,
color: Color | str = WHITE,
**kwargs,
):
    super().__init__(
        arc_center=point,
        radius=radius,
        stroke_width=stroke_width,
        fill_opacity=fill_opacity,
        color=color,
        **kwargs,
    )

class AnnotationDot(Dot):
    """A dot with bigger radius and bold stroke to annotate scenes."""

    def __init__(self,
                 radius: float = DEFAULT_DOT_RADIUS * 1.3,
                 stroke_width=5,
                 stroke_color=WHITE,
                 fill_color=BLUE,
                 **kwargs,
                 ):
        super().__init__(
            radius=radius,
            stroke_width=stroke_width,
            stroke_color=stroke_color,
            fill_color=fill_color,
            **kwargs,
        )

class LabeledDot(Dot):
    """A :class:`Dot` containing a label in its center.

    Parameters
    -----
    label
        The label of the :class:`Dot`. This is rendered as :class:`~.MathTex`
        by default (i.e., when passing a :class:`str`), but other classes
        representing rendered strings like :class:`~.Text` or :class:`~.Tex`
        can be passed as well.
    radius
        The radius of the :class:`Dot`. If ``None`` (the default), the radius
        is calculated based on the size of the ``label``.

    Examples
    -----
    .. manim:: SeveralLabeledDots
       :save_last_frame:

       class SeveralLabeledDots(Scene):
           def construct(self):
               sq = Square(fill_color=RED, fill_opacity=1)
               self.add(sq)
               dot1 = LabeledDot(Tex("42", color=RED))
               dot2 = LabeledDot(MathTex("a", color=GREEN))
               dot3 = LabeledDot(Text("ii", color=BLUE))
               dot4 = LabeledDot("3")
               dot1.next_to(sq, UL)
               dot2.next_to(sq, UR)
               dot3.next_to(sq, DL)
               dot4.next_to(sq, DR)
               self.add(dot1, dot2, dot3, dot4)
       """

    def __init__(self,
                 label: str | SingleStringMathTex | Text | Tex,
                 radius: float | None = None,
                 **kwargs,
                 ) -> None:
        if isinstance(label, str):
            from manim import MathTex
            rendered_label = MathTex(label, color=BLACK)
        else:
            rendered_label = label

        if radius is None:
            radius = 0.1 + max(rendered_label.width, rendered_label.height) / 2
        super().__init__(radius=radius, **kwargs)
        rendered_label.move_to(self.get_center())
        self.add(rendered_label)

class Ellipse(Circle):
    """A circular shape; oval, circle.

    Parameters
    -----
    width
        The horizontal width of the ellipse.

```

```

height
    The vertical height of the ellipse.
kwargs
    Additional arguments to be passed to :class:`Circle`.

Examples
-----
.. manim:: EllipseExample
:save_last_frame:

class EllipseExample(Scene):
    def construct(self):
        ellipse_1 = Ellipse(width=2.0, height=4.0, color=BLUE_B)
        ellipse_2 = Ellipse(width=4.0, height=1.0, color=BLUE_D)
        ellipse_group = Group(ellipse_1, ellipse_2).arrange(buff=1)
        self.add(ellipse_group)

def __init__(self, width: float = 2, height: float = 1, **kwargs):
    super().__init__(**kwargs)
    self.stretch_to_fit_width(width)
    self.stretch_to_fit_height(height)

class AnnularSector(Arc):
    """A sector of an annulus.

Parameters
-----
inner_radius
    The inside radius of the Annular Sector.
outer_radius
    The outside radius of the Annular Sector.
angle
    The clockwise angle of the Annular Sector.
start_angle
    The starting clockwise angle of the Annular Sector.
fill_opacity
    The opacity of the color filled in the Annular Sector.
stroke_width
    The stroke width of the Annular Sector.
color
    The color filled into the Annular Sector.

Examples
-----
.. manim:: AnnularSectorExample
:save_last_frame:

class AnnularSectorExample(Scene):
    def construct(self):
        # Changes background color to clearly visualize changes in fill_opacity.
        self.camera.background_color = WHITE

        # The default parameter start_angle is 0, so the AnnularSector starts from the +x-axis.
        s1 = AnnularSector(color=YELLOW).move_to(2 * UL)

        # Different inner_radius and outer_radius than the default.
        s2 = AnnularSector(inner_radius=1.5, outer_radius=2, angle=45 * DEGREES, color=RED).move_to(2 * UR)

        # fill_opacity is typically a number > 0 and <= 1. If fill_opacity=0, the AnnularSector is transparent.
        s3 = AnnularSector(inner_radius=1, outer_radius=1.5, angle=PI, fill_opacity=0.25, color=BLUE).move_to(2 * DL)

        # With a negative value for the angle, the AnnularSector is drawn clockwise from the start value.
        s4 = AnnularSector(inner_radius=1, outer_radius=1.5, angle=-3 * PI / 2, color=GREEN).move_to(2 * DR)

        self.add(s1, s2, s3, s4)

    def __init__(
        self,
        inner_radius=1,
        outer_radius=2,
        angle=TAU / 4,
        start_angle=0,
        fill_opacity=1,
        stroke_width=0,
        color=WHITE,
        **kwargs,
    ):
        self.inner_radius = inner_radius
        self.outer_radius = outer_radius
        super().__init__(
            start_angle=start_angle,
            angle=angle,
            fill_opacity=fill_opacity,
            stroke_width=stroke_width,
            color=color,
            **kwargs,
        )

    def generate_points(self):
        inner_arc, outer_arc = (
            Arc(

```

```

        start_angle=self.start_angle,
        angle=self.angle,
        radius=radius,
        arc_center=self.arc_center,
    )
    for radius in (self.inner_radius, self.outer_radius)
)
outer_arc.reverse_points()
self.append_points(inner_arc.points)
self.add_line_to(outer_arc.points[0])
self.append_points(outer_arc.points)
self.add_line_to(inner_arc.points[0])

init_points = generate_points

class Sector(AnnularSector):
    """A sector of a circle.

Examples
-----
.. manim:: ExampleSector
:save_last_frame:

class ExampleSector(Scene):
def construct(self):
    sector = Sector(outer_radius=2, inner_radius=1)
    sector2 = Sector(outer_radius=2.5, inner_radius=0.8).move_to([-3, 0, 0])
    sector.set_color(RED)
    sector2.set_color(PINK)
    self.add(sector, sector2)
"""

def __init__(self, outer_radius=1, inner_radius=0, **kwargs):
    super().__init__(inner_radius=inner_radius, outer_radius=outer_radius, **kwargs)

class Annulus(Circle):
    """Region between two concentric :class:`Circles <.Circle>`.

Parameters
-----
inner_radius
    The radius of the inner :class:`Circle`.
outer_radius
    The radius of the outer :class:`Circle`.
kwargs
    Additional arguments to be passed to :class:`Annulus`


Examples
-----
.. manim:: AnnulusExample
:save_last_frame:

class AnnulusExample(Scene):
def construct(self):
    annulus_1 = Annulus(inner_radius=0.5, outer_radius=1).shift(UP)
    annulus_2 = Annulus(inner_radius=0.3, outer_radius=0.6, color=RED).next_to(annulus_1, DOWN)
    self.add(annulus_1, annulus_2)
"""

def __init__(
    self,
    inner_radius: float | None = 1,
    outer_radius: float | None = 2,
    fill_opacity=1,
    stroke_width=0,
    color=WHITE,
    mark_paths_closed=False,
    **kwargs,
):
    self.mark_paths_closed = mark_paths_closed # is this even used?
    self.inner_radius = inner_radius
    self.outer_radius = outer_radius
    super().__init__(
        fill_opacity=fill_opacity, stroke_width=stroke_width, color=color, **kwargs
    )

def generate_points(self):
    self.radius = self.outer_radius
    outer_circle = Circle(radius=self.outer_radius)
    inner_circle = Circle(radius=self.inner_radius)
    inner_circle.reverse_points()
    self.append_points(outer_circle.points)
    self.append_points(inner_circle.points)
    self.shift(self.arc_center)

init_points = generate_points

class CubicBezier(VMobject, metaclass=ConvertToOpenGL):
    """A cubic Bézier curve.

Example
-----

```



```

ap_group = VGroup(ap1, ap2, ap3, ap4).arrange()
self.play(*[Create(ap) for ap in [ap1, ap2, ap3, ap4]])
self.wait()

For further examples see :class:`ArcPolygonFromArcs`.

"""

def __init__(self,
             *vertices: list | np.ndarray,
             angle: float = PI / 4,
             radius: float | None = None,
             arc_config: list[dict] | None = None,
             **kwargs,
             ):
    n = len(vertices)
    point_pairs = [(vertices[k], vertices[(k + 1) % n]) for k in range(n)]

    if not arc_config:
        if radius:
            all_arc_configs = itertools.repeat({"radius": radius}, len(point_pairs))
        else:
            all_arc_configs = itertools.repeat({"angle": angle}, len(point_pairs))
    elif isinstance(arc_config, dict):
        all_arc_configs = itertools.repeat(arc_config, len(point_pairs))
    else:
        assert len(arc_config) == n
        all_arc_configs = arc_config

    arcs = [
        ArcBetweenPoints(*pair, **conf)
        for (pair, conf) in zip(point_pairs, all_arc_configs)
    ]

    super().__init__(**kwargs)
    # Adding the arcs like this makes ArcPolygon double as a VGroup.
    # Also makes changes to the ArcPolygon, such as scaling, affect
    # the arcs, so that their new values are usable.
    self.add(*arcs)
    for arc in arcs:
        self.append_points(arc.points)

    # This enables the use of ArcPolygon.arcs as a convenience
    # because ArcPolygon[0] returns itself, not the first Arc.
    self.arcs = arcs

```

```

class ArcPolygonFromArcs(VMobject, metaclass=ConvertToOpenGL):
    """A generalized polygon allowing for points to be connected with arcs.

```

This version takes in pre-defined arcs to generate the arcpolygon and introduces little new syntax. However unlike :class:`Polygon` it can't be created with points directly.

For proper appearance the passed arcs should connect seamlessly:
``[a,b][b,c][c,a]``

If there are any gaps between the arcs, those will be filled in with straight lines, which can be used deliberately for any straight sections. Arcs can also be passed as straight lines such as an arc initialized with ``angle=0``.

Parameters

arcs

These are the arcs from which the arcpolygon is assembled.

kwargs

Keyword arguments that are passed to the constructor of :class:`~.VMobject`. Affects how the ArcPolygon itself is drawn, but doesn't affect passed arcs.

Attributes

arcs

The arcs used to initialize the ArcPolygonFromArcs::

```

>>> from manim import ArcPolygonFromArcs, Arc, ArcBetweenPoints
>>> ap = ArcPolygonFromArcs(Arc(), ArcBetweenPoints([1,0,0], [0,1,0]), Arc())
>>> ap.arcs
[Arc, ArcBetweenPoints, Arc]

```

.. tip::

Two instances of :class:`ArcPolygon` can be transformed properly into one another as well. Be advised that any arc initialized with ``angle=0`` will actually be a straight line, so if a straight section should seamlessly transform into an arced section or vice versa, initialize the straight section with a negligible angle instead (such as ``angle=0.0001``).

.. note::

There is an alternative version (:class:`ArcPolygon`) that can be instantiated with points.

.. seealso::

```
:class:`ArcPolygon`
```

Examples

One example of an arcpolygon is the Reuleaux triangle.
Instead of 3 straight lines connecting the outer points,
a Reuleaux triangle has 3 arcs connecting those points,
making a shape with constant width.

Passed arcs are stored as subobjects in the arcpolygon.
This means that the arcs are changed along with the arcpolygon,
for example when it's shifted, and these arcs can be manipulated
after the arcpolygon has been initialized.

Also both the arcs contained in an :class:`~.ArcPolygonFromArcs`, as well as the
arcpolygon itself are drawn, which affects draw time in :class:`~.Create`
for example. In most cases the arcs themselves don't
need to be drawn, in which case they can be passed as invisible.

```
.. manim:: ArcPolygonExample
```

```
class ArcPolygonExample(Scene):
    def construct(self):
        arc_conf = {"stroke_width": 0}
        poly_conf = {"stroke_width": 10, "stroke_color": BLUE,
                    "fill_opacity": 1, "color": PURPLE}
        a = [-1, 0, 0]
        b = [1, 0, 0]
        c = [0, np.sqrt(3), 0]
        arc0 = ArcBetweenPoints(a, b, radius=2, **arc_conf)
        arc1 = ArcBetweenPoints(b, c, radius=2, **arc_conf)
        arc2 = ArcBetweenPoints(c, a, radius=2, **arc_conf)
        reuleaux_tri = ArcPolygonFromArcs(arc0, arc1, arc2, **poly_conf)
        self.play(FadeIn(reuleaux_tri))
        self.wait(2)
```

The arcpolygon itself can also be hidden so that instead only the contained
arcs are drawn. This can be used to easily debug arcs or to highlight them.

```
.. manim:: ArcPolygonExample2
```

```
class ArcPolygonExample2(Scene):
    def construct(self):
        arc_conf = {"stroke_width": 3, "stroke_color": BLUE,
                    "fill_opacity": 0.5, "color": GREEN}
        poly_conf = {"color": None}
        a = [-1, 0, 0]
        b = [1, 0, 0]
        c = [0, np.sqrt(3), 0]
        arc0 = ArcBetweenPoints(a, b, radius=2, **arc_conf)
        arc1 = ArcBetweenPoints(b, c, radius=2, **arc_conf)
        arc2 = ArcBetweenPoints(c, a, radius=2, stroke_color=RED)
        reuleaux_tri = ArcPolygonFromArcs(arc0, arc1, arc2, **poly_conf)
        self.play(FadeIn(reuleaux_tri))
        self.wait(2)

def __init__(self, *arcs: Arc | ArcBetweenPoints, **kwargs):
    if not all(isinstance(m, (Arc, ArcBetweenPoints)) for m in arcs):
        raise ValueError(
            "All ArcPolygon subobjects must be of type Arc/ArcBetweenPoints",
        )
    super().__init__(**kwargs)
    # Adding the arcs like this makes ArcPolygonFromArcs double as a VGroup.
    # Also makes changes to the ArcPolygonFromArcs, such as scaling, affect
    # the arcs, so that their new values are usable.
    self.add(*arcs)
    # This enables the use of ArcPolygonFromArcs.arcs as a convenience
    # because ArcPolygonFromArcs[0] returns itself, not the first Arc.
    self.arcs = [*arcs]
    from .line import Line

    for arc1, arc2 in adjacent_pairs(arcs):
        self.append_points(arc1.points)
        line = Line(arc1.get_end(), arc2.get_start())
        len_ratio = line.get_length() / arc1.get_arc_length()
        if math.isnan(len_ratio) or math.isinf(len_ratio):
            continue
        line.insert_n_curves(int(arc1.get_num_curves() * len_ratio))
        self.append_points(line.points)
```

opengl_point_cloud_mobject.py

```
from __future__ import annotations

__all__ = ["OpenGLPObject", "OpenGLGroup", "OpenGLPMPoint"]

import moderngl
import numpy as np

from manim.constants import *
from manim.mobject.opengl_mobject import OpenGLMObject
from manim.utils.bezier import interpolate
from manim.utils.color import BLACK, WHITE, YELLOW, color_gradient, color_to_rgba
from manim.utils.config_ops import _Uniforms
from manim.utils.iterables import resize_with_interpolation

class OpenGLPObject(OpenGLMObject):
    shader_folder = "true_dot"
    # Scale for consistency with cairo units
    OPENGL_POINT_RADIUS_SCALE_FACTOR = 0.01
    shader_dtype = [
        ("point", np.float32, (3,)),
        ("color", np.float32, (4,)),
    ]
    point_radius = _Uniforms()

    def __init__(self, stroke_width=2.0, color=YELLOW, render_primitive=moderngl.POINTS, **kwargs):
        self.stroke_width = stroke_width
        super().__init__(color=color, render_primitive=render_primitive, **kwargs)
        self.point_radius = (
            self.stroke_width * OpenGLPObject.OPENGL_POINT_RADIUS_SCALE_FACTOR
        )

    def reset_points(self):
        self.rgbas = np.zeros((1, 4))
        self.points = np.zeros((0, 3))
        return self

    def get_array_attrs(self):
        return ["points", "rgbas"]

    def add_points(self, points, rgbas=None, color=None, opacity=None):
        """Add points.

        Points must be a Nx3 numpy array.
        Rgbas must be a Nx4 numpy array if it is not None.
        """
        if rgbas is None and color is None:
            color = YELLOW
        self.append_points(points)
        # Rgbas array will have been resized with points
        if color is not None:
            if opacity is None:
                opacity = self.rgbas[-1, 3]
            new_rgbas = np.repeat([color_to_rgba(color, opacity)], len(points), axis=0)
        elif rgbas is not None:
            new_rgbas = rgbas
        elif len(rgbas) != len(points):
            raise ValueError("points and rgbas must have same length")
        self.rgbas = np.append(self.rgbas, new_rgbas, axis=0)
        return self

    def thin_out(self, factor=5):
        """Removes all but every nth point for n = factor"""
        for mob in self.family_members_with_points():
            num_points = mob.get_num_points()

            def thin_func():
                return np.arange(0, num_points, factor)

            if len(mob.points) == len(mob.rgbas):
                mob.set_rgba_array_direct(mob.rgbas[thin_func()])
                mob.set_points(mob.points[thin_func()])

        return self

    def set_color_by_gradient(self, *colors):
```

```

self.rgbas = np.array(
    list(map(color_to_rgba, color_gradient(*colors, self.get_num_points()))),
)
return self

def set_colors_by_radial_gradient(
    self,
    center=None,
    radius=1,
    inner_color=WHITE,
    outer_color=BLACK,
):
    start_rgba, end_rgba = list(map(color_to_rgba, [inner_color, outer_color]))
    if center is None:
        center = self.get_center()
    for mob in self.family_members_with_points():
        distances = np.abs(self.points - center)
        alphas = np.linalg.norm(distances, axis=1) / radius

        mob.rgbas = np.array(
            np.array(
                [interpolate(start_rgba, end_rgba, alpha) for alpha in alphas],
            ),
        )
    return self

def match_colors(self, pmobject):
    self.rgbas[:] = resize_with_interpolation(pmobject.rgbas, self.get_num_points())
    return self

def fade_to(self, color, alpha, family=True):
    rgbas = interpolate(self.rgbas, color_to_rgba(color), alpha)
    for mob in self.subobjects:
        mob.fade_to(color, alpha, family)
    self.set_rgba_array_direct(rgbas)
    return self

def filter_out(self, condition):
    for mob in self.family_members_with_points():
        to_keep = ~np.apply_along_axis(condition, 1, mob.points)
        for key in mob.data:
            mob.data[key] = mob.data[key][to_keep]
    return self

def sort_points(self, function=lambda p: p[0]):
    """
    function is any map from R^3 to R
    """
    for mob in self.family_members_with_points():
        indices = np.argsort(np.apply_along_axis(function, 1, mob.points))
        for key in mob.data:
            mob.data[key] = mob.data[key][indices]
    return self

def ingest_subobjects(self):
    for key in self.data:
        self.data[key] = np.vstack([sm.data[key] for sm in self.get_family()])
    return self

def point_from_proportion(self, alpha):
    index = alpha * (self.get_num_points() - 1)
    return self.points[int(index)]

def pointwise_become_partial(self, pmobject, a, b):
    lower_index = int(a * pmobject.get_num_points())
    upper_index = int(b * pmobject.get_num_points())
    for key in self.data:
        self.data[key] = pmobject.data[key][lower_index:upper_index]
    return self

def get_shader_data(self):
    shader_data = np.zeros(len(self.points), dtype=self.shader_dtype)
    self.read_data_to_shader(shader_data, "point", "points")
    self.read_data_to_shader(shader_data, "color", "rgbas")
    return shader_data

@staticmethod
def get_mobject_type_class():
    return OpenGLMObject

class OpenGLGroup(OpenGLMObject):
    def __init__(self, *pmobs, **kwargs):
        if not all(isinstance(m, OpenGLMObject) for m in pmobs):
            raise Exception("All subobjects must be of type OpenGLMObject")
        super().__init__(**kwargs)
        self.add(*pmobs)

```

```
def fade_to(self, color, alpha, family=True):
    if family:
        for mob in self.submobjects:
            mob.fade_to(color, alpha, family)

class OpenGLPMPPoint(OpenGLPMPObject):
    def __init__(self, location=ORIGIN, stroke_width=4.0, **kwargs):
        self.location = location
        super().__init__(stroke_width=stroke_width, **kwargs)

    def init_points(self):
        self.points = np.array([self.location], dtype=np.float32)
```

specialized.py

```
from __future__ import annotations
__all__ = ["Broadcast"]
from typing import Any, Sequence
from manim.animation.transform import Restore
from ..constants import *
from .composition import LaggedStart

class Broadcast(LaggedStart):
    """Broadcast a mobject starting from an ``initial_width``, up to the actual size of the mobject.

    Parameters
    -----
    mobject
        The mobject to be broadcast.
    focal_point
        The center of the broadcast, by default ORIGIN.
    n_mobs
        The number of mobjects that emerge from the focal point, by default 5.
    initial_opacity
        The starting stroke opacity of the mobjects emitted from the broadcast, by default 1.
    final_opacity
        The final stroke opacity of the mobjects emitted from the broadcast, by default 0.
    initial_width
        The initial width of the mobjects, by default 0.0.
    remover
        Whether the mobjects should be removed from the scene after the animation, by default True.
    lag_ratio
        The time between each iteration of the mobject, by default 0.2.
    run_time
        The total duration of the animation, by default 3.
    kwargs
        Additional arguments to be passed to :class:`~.LaggedStart`.

    Examples
    -----
    .. manim:: BroadcastExample

        class BroadcastExample(Scene):
            def construct(self):
                mob = Circle(radius=4, color=TEAL_A)
                self.play(Broadcast(mob))
    """

    def __init__(
        self,
        mobject,
        focal_point: Sequence[float] = ORIGIN,
        n_mobs: int = 5,
        initial_opacity: float = 1,
        final_opacity: float = 0,
        initial_width: float = 0.0,
        remover: bool = True,
        lag_ratio: float = 0.2,
        run_time: float = 3,
        **kwargs: Any,
    ):
        self.focal_point = focal_point
        self.n_mobs = n_mobs
        self.initial_opacity = initial_opacity
        self.final_opacity = final_opacity
        self.initial_width = initial_width

        anims = []

        # Works by saving the mob that is passed into the animation, scaling it to 0 (or the initial_width) and then restoring the original mob.
        if mobject.fill_opacity:
            fill_o = True
        else:
            fill_o = False

        for _ in range(self.n_mobs):
            mob = mobject.copy()

            if fill_o:
                mob.set_opacity(self.final_opacity)
            else:
                mob.set_stroke(opacity=self.final_opacity)

            mob.move_to(self.focal_point)
            mob.save_state()
            mob.set_width(self.initial_width)

            if fill_o:
                mob.set_opacity(self.initial_opacity)
            else:
                mob.set_stroke(opacity=self.initial_opacity)

            anims.append(Restore(mob, remover))

    super().__init__(*anims, run_time=run_time, lag_ratio=lag_ratio, **kwargs)
```

animation.py

```
"""Animate mobjects."""

from __future__ import annotations

from manim.mobject.opengl.opengl_mobject import OpenGLMObject

from .. import config, logger
from ..constants import RendererType
from ..mobject import mobject
from ..mobject.mobject import Mobject
from ..mobject.opengl import opengl_mobject
from ..utils.rate_functions import linear, smooth

__all__ = ["Animation", "Wait", "override_animation"]

from copy import deepcopy
from typing import TYPE_CHECKING, Callable, Iterable, Sequence

if TYPE_CHECKING:
    from manim.scene.scene import Scene

DEFAULT_ANIMATION_RUN_TIME: float = 1.0
DEFAULT_ANIMATION_LAG_RATIO: float = 0.0

class Animation:
    """An animation.

    Animations have a fixed time span.

    Parameters
    -----
    mobject
        The mobject to be animated. This is not required for all types of animations.
    lag_ratio
        Defines the delay after which the animation is applied to submobjects. This lag
        is relative to the duration of the animation.

        This does not influence the total
        runtime of the animation. Instead the runtime of individual animations is
        adjusted so that the complete animation has the defined run time.

    run_time
        The duration of the animation in seconds.
    rate_func
        The function defining the animation progress based on the relative runtime (see :mod:`~.rate_functions`).

        For example ``rate_func(0.5)`` is the proportion of the animation that is done
        after half of the animations run time.

    reverse_rate_function
        Reverses the rate function of the animation. Setting ``reverse_rate_function``
        does not have any effect on ``remover`` or ``introducer``. These need to be
        set explicitly if an introducer-animation should be turned into a remover one
        and vice versa.
    name
        The name of the animation. This gets displayed while rendering the animation.
        Defaults to <class-name>(<Mobject-name>).
    remover
        Whether the given mobject should be removed from the scene after this animation.
    suspend_mobject_updating
        Whether updaters of the mobject should be suspended during the animation.

    .. NOTE::

        In the current implementation of this class, the specified rate function is applied
        within :meth:`.Animation.interpolate_mobject` call as part of the call to
        :meth:`.Animation.interpolate_submobject`. For subclasses of :class:`.Animation`
        that are implemented by overriding :meth:`.interpolate_mobject`, the rate function
        has to be applied manually (e.g., by passing ``self.rate_func(alpha)`` instead
        of just ``alpha``).

    Examples
    -----
    .. manim:: LagRatios
```

```

class LagRatios(Scene):
    def construct(self):
        ratios = [0, 0.1, 0.5, 1, 2] # demonstrated lag_ratios

        # Create dot groups
        group = VGroup(*[Dot() for _ in range(4)]).arrange_submobjects()
        groups = VGroup(*[group.copy() for _ in ratios]).arrange_submobjects(buff=1)
        self.add(groups)

        # Label groups
        self.add(Text("lag_ratio = ", font_size=36).next_to(groups, UP, buff=1.5))
        for group, ratio in zip(groups, ratios):
            self.add(Text(str(ratio), font_size=36).next_to(group, UP))

        #Animate groups with different lag_ratios
        self.play(AnimationGroup(*[
            group.animate(lag_ratio=ratio, run_time=1.5).shift(DOWN * 2)
            for group, ratio in zip(groups, ratios)
        ]))

        # lag_ratio also works recursively on nested submobjects:
        self.play(groups.animate(run_time=1, lag_ratio=0.1).shift(UP * 2))

"""

def __new__(cls,
          mobject=None,
          *args,
          use_override=True,
          **kwargs,
          ):
    if isinstance(mobject, Mobject) and use_override:
        func = mobject.animation_override_for(cls)
        if func is not None:
            anim = func(mobject, *args, **kwargs)
            logger.debug(
                f"The {cls.__name__} animation has been overridden for "
                f"{type(mobject).__name__} mobjects. use_override = False can "
                f"be used as keyword argument to prevent animation overriding.",
            )
        return anim
    return super().__new__(cls)

def __init__(self,
            mobject: Mobject | None,
            lag_ratio: float = DEFAULT_ANIMATION_LAG_RATIO,
            run_time: float = DEFAULT_ANIMATION_RUN_TIME,
            rate_func: Callable[[float], float] = smooth,
            reverse_rate_function: bool = False,
            name: str = None,
            remover: bool = False, # remove a mobject from the screen?
            suspend_mobject_updating: bool = True,
            introducer: bool = False,
            *,
            _on_finish: Callable[[], None] = lambda _: None,
            **kwargs,
            ) -> None:
    self._typecheck_input(mobject)
    self.run_time: float = run_time
    self.rate_func: Callable[[float], float] = rate_func
    self.reverse_rate_function: bool = reverse_rate_function
    self.name: str | None = name
    self.remover: bool = remover
    self.introducer: bool = introducer
    self.suspend_mobject_updating: bool = suspend_mobject_updating
    self.lag_ratio: float = lag_ratio
    self._on_finish: Callable[[Scene], None] = _on_finish
    if config["renderer"] == RendererType.OPENGL:
        self.starting_mobject: OpenGLMobject = OpenGLMobject()
        self.mobject: OpenGLMobject = (
            mobject if mobject is not None else OpenGLMobject()
        )
    else:
        self.starting_mobject: Mobject = Mobject()
        self.mobject: Mobject = mobject if mobject is not None else Mobject()
    if kwargs:
        logger.debug("Animation received extra kwargs: %s", kwargs)

    if hasattr(self, "CONFIG"):
        logger.error(
            (
                "CONFIG has been removed from ManimCommunity.",
                "Please use keyword arguments instead.",
            ),
        )


```

```

def _typecheck_input(self, mobject: Mobject | None) -> None:
    if mobject is None:
        logger.debug("Animation with empty mobject")
    elif not isinstance(mobject, (Mobject, OpenGLMobject)):
        raise TypeError("Animation only works on Mobjects")

def __str__(self) -> str:
    if self.name:
        return self.name
    return f"{self.__class__.__name__}({str(self.mobject)})"

def __repr__(self) -> str:
    return str(self)

def begin(self) -> None:
    """Begin the animation.

    This method is called right as an animation is being played. As much
    initialization as possible, especially any mobject copying, should live in this
    method.

    """
    self.starting_mobject = self.create_starting_mobject()
    if self.suspend_mobject_updating:
        # All calls to self.mobject's internal updaters
        # during the animation, either from this Animation
        # or from the surrounding scene, should do nothing.
        # It is, however, okay and desirable to call
        # the internal updaters of self.starting_mobject,
        # or any others among self.get_all_mobjects()
        self.mobject.suspend_updating()
    self.interpolate(0)

def finish(self) -> None:
    # TODO: begin and finish should require a scene as parameter.
    # That way Animation.clean_up_from_screen and Scene.add_mobjects_from_animations
    # could be removed as they fulfill basically the same purpose.
    """Finish the animation.

    This method gets called when the animation is over.

    """
    self.interpolate(1)
    if self.suspend_mobject_updating and self.mobject is not None:
        self.mobject.resume_updating()

def clean_up_from_scene(self, scene: Scene) -> None:
    """Clean up the :class:`~.Scene` after finishing the animation.

    This includes to :meth:`~.Scene.remove` the Animation's
    :class:`~.Mobject` if the animation is a remover.

    Parameters
    -----
    scene
        The scene the animation should be cleaned up from.

    """
    self._on_finish(scene)
    if self.is_remover():
        scene.remove(self.mobject)

def setup_scene(self, scene: Scene) -> None:
    """Setup up the :class:`~.Scene` before starting the animation.

    This includes to :meth:`~.Scene.add` the Animation's
    :class:`~.Mobject` if the animation is an introducer.

    Parameters
    -----
    scene
        The scene the animation should be cleaned up from.

    """
    if scene is None:
        return
    if (
        self.is_introducer()
        and self.mobject not in scene.get_mobject_family_members()
    ):
        scene.add(self.mobject)

def create_starting_mobject(self) -> Mobject:
    # Keep track of where the mobject starts
    return self.mobject.copy()

def get_all_mobjects(self) -> Sequence[Mobject]:
    """Get all mobjects involved in the animation.

```

```

Ordering must match the ordering of arguments to interpolate_submobject

Returns
-----
Sequence[Mobject]
    The sequence of mobjects.
"""
return self.mobject, self.starting_mobject

def get_all_families_zipped(self) -> Iterable[tuple]:
    if config["renderer"] == RendererType.OPENGL:
        return zip(*[mob.get_family() for mob in self.get_all_mobjects()])
    return zip(
        *(mob.family_members_with_points() for mob in self.get_all_mobjects())
    )

def update_mobjects(self, dt: float) -> None:
    """
    Updates things like starting_mobject, and (for
    transforms) target_mobject. Note, since typically
    (always?) self.mobject will have its updating
    suspended during the animation, this will do
    nothing to self.mobject.
    """
    for mob in self.get_all_mobjects_to_update():
        mob.update(dt)

def get_all_mobjects_to_update(self) -> list[Mobject]:
    """Get all mobjects to be updated during the animation.

    Returns
    -----
    List[Mobject]
        The list of mobjects to be updated during the animation.
    """
    # The surrounding scene typically handles
    # updating of self.mobject. Besides, in
    # most cases its updating is suspended anyway
    return list(filter(lambda m: m is not self.mobject, self.get_all_mobjects()))

def copy(self) -> Animation:
    """Create a copy of the animation.

    Returns
    -----
    Animation
        A copy of ``self``.
    """
    return deepcopy(self)

# Methods for interpolation, the mean of an Animation

# TODO: stop using alpha as parameter name in different meanings.
def interpolate(self, alpha: float) -> None:
    """Set the animation progress.

    This method gets called for every frame during an animation.

    Parameters
    -----
    alpha
        The relative time to set the animation to, 0 meaning the start, 1 meaning
        the end.
    """
    self.interpolate_mobject(alpha)

def interpolate_mobject(self, alpha: float) -> None:
    """Interpolates the mobject of the :class:`Animation` based on alpha value.

    Parameters
    -----
    alpha
        A float between 0 and 1 expressing the ratio to which the animation
        is completed. For example, alpha-values of 0, 0.5, and 1 correspond
        to the animation being completed 0%, 50%, and 100%, respectively.
    """
    families = list(self.get_all_families_zipped())
    for i, mobs in enumerate(families):
        sub_alpha = self.get_sub_alpha(alpha, i, len(families))
        self.interpolate_submobject(*mobs, sub_alpha)

def interpolate_submobject(
    self,
    submobject: Mobject,
    starting_submobject: Mobject,
    # target_copy: Mobject, #Todo: fix - signature of interpolate_submobject differs in Transform().
    alpha: float,
) -> Animation:

```

```

# Typically implemented by subclass
pass

def get_sub_alpha(self, alpha: float, index: int, num_subobjects: int) -> float:
    """Get the animation progress of any subobjects subanimation.

    Parameters
    -----
    alpha
        The overall animation progress
    index
        The index of the subanimation.
    num_subobjects
        The total count of subanimations.

    Returns
    -----
    float
        The progress of the subanimation.
    """
    # TODO, make this more understandable, and/or combine
    # its functionality with AnimationGroup's method
    # build_animations_with_timings
    lag_ratio = self.lag_ratio
    full_length = (num_subobjects - 1) * lag_ratio + 1
    value = alpha * full_length
    lower = index * lag_ratio
    if self.reverse_rate_function:
        return self.rate_func(1 - (value - lower))
    else:
        return self.rate_func(value - lower)

# Getters and setters
def set_run_time(self, run_time: float) -> Animation:
    """Set the run time of the animation.

    Parameters
    -----
    run_time
        The new time the animation should take in seconds.

    .. note::
        The run_time of an animation should not be changed while it is already
        running.

    Returns
    -----
    Animation
        ``self``
    """
    self.run_time = run_time
    return self

def get_run_time(self) -> float:
    """Get the run time of the animation.

    Returns
    -----
    float
        The time the animation takes in seconds.
    """
    return self.run_time

def set_rate_func(
    self,
    rate_func: Callable[[float], float],
) -> Animation:
    """Set the rate function of the animation.

    Parameters
    -----
    rate_func
        The new function defining the animation progress based on the
        relative runtime (see :mod:`~.rate_functions`).

    Returns
    -----
    Animation
        ``self``
    """
    self.rate_func = rate_func
    return self

def get_rate_func(
    self,
) -> Callable[[float], float]:
    """Get the rate function of the animation.

```

```

    Returns
    -----
    Callable[[float], float]
        The rate function of the animation.
    """
    return self.rate_func

def set_name(self, name: str) -> Animation:
    """Set the name of the animation.

    Parameters
    -----
    name
        The new name of the animation.

    Returns
    -----
    Animation
        ``self``

    """
    self.name = name
    return self

def is_remover(self) -> bool:
    """Test if the animation is a remover.

    Returns
    -----
    bool
        ``True`` if the animation is a remover, ``False`` otherwise.

    """
    return self.remover

def is_introducer(self) -> bool:
    """Test if the animation is an introducer.

    Returns
    -----
    bool
        ``True`` if the animation is an introducer, ``False`` otherwise.

    """
    return self.introducer

def prepare_animation(
    anim: Animation | mobject._AnimationBuilder,
) -> Animation:
    """Returns either an unchanged animation, or the animation built
    from a passed animation factory.

    Examples
    -----
    ::

        >>> from manim import Square, FadeIn
        >>> s = Square()
        >>> prepare_animation(FadeIn(s))
        FadeIn(Square)

    ::

        >>> prepare_animation(s.animate.scale(2).rotate(42))
        _MethodAnimation(Square)

    ::

        >>> prepare_animation(42)
        Traceback (most recent call last):
        ...
        TypeError: Object 42 cannot be converted to an animation

    """
    if isinstance(anim, mobject._AnimationBuilder):
        return anim.build()

    if isinstance(anim, opengl_mobject._AnimationBuilder):
        return anim.build()

    if isinstance(anim, Animation):
        return anim

    raise TypeError(f"Object {anim} cannot be converted to an animation")

class Wait(Animation):
    """A "no operation" animation.

```

```

Parameters
-----
run_time
    The amount of time that should pass.
stop_condition
    A function without positional arguments that evaluates to a boolean.
    The function is evaluated after every new frame has been rendered.
    Playing the animation stops after the return value is truthy, or
    after the specified ``run_time`` has passed.
frozen_frame
    Controls whether or not the wait animation is static, i.e., corresponds
    to a frozen frame. If ``False`` is passed, the render loop still
    progresses through the animation as usual and (among other things)
    continues to call updater functions. If ``None`` (the default value),
    the :meth:`Scene.play` call tries to determine whether the Wait call
    can be static or not itself via :meth:`Scene.should_mobjects_update`.
kwargs
    Keyword arguments to be passed to the parent class, :class:`.Animation`.
"""

def __init__(
    self,
    run_time: float = 1,
    stop_condition: Callable[[], bool] | None = None,
    frozen_frame: bool | None = None,
    rate_func: Callable[[float], float] = linear,
    **kwargs,
):
    if stop_condition and frozen_frame:
        raise ValueError("A static Wait animation cannot have a stop condition.")

    self.duration: float = run_time
    self.stop_condition = stop_condition
    self.is_static_wait: bool = frozen_frame
    super().__init__(None, run_time=run_time, rate_func=rate_func, **kwargs)
    # quick fix to work in opengl setting:
    self.mobject.shader_wrapper_list = []

def begin(self) -> None:
    pass

def finish(self) -> None:
    pass

def clean_up_from_scene(self, scene: Scene) -> None:
    pass

def update_mobjects(self, dt: float) -> None:
    pass

def interpolate(self, alpha: float) -> None:
    pass

def override_animation(
    animation_class: type[Animation],
) -> Callable[[Callable], Callable]:
    """Decorator used to mark methods as overrides for specific :class:`~.Animation` types.

    Should only be used to decorate methods of classes derived from :class:`~.Mobject`.
    ``Animation`` overrides get inherited to subclasses of the ``Mobject`` who defined
    them. They don't override subclasses of the ``Animation`` they override.

    See Also
    -----
    :meth:`~.Mobject.add_animation_override`

    Parameters
    -----
    animation_class
        The animation to be overridden.

    Returns
    -----
    Callable[[Callable], Callable]
        The actual decorator. This marks the method as overriding an animation.

    Examples
    -----
    .. manim:: OverrideAnimationExample

        class MySquare(Square):
            @override_animation(FadeIn)
            def _fade_in_override(self, **kwargs):
                return Create(self, **kwargs)
    """

```

```
class OverrideAnimationExample(Scene):
    def construct(self):
        self.play(FadeIn(MySquare()))


def decorator(func):
    func._override_animation = animation_class
    return func

return decorator
```

fading.py

```
"""Fading in and out of view.

.. manim:: Fading

    class Fading(Scene):
        def construct(self):
            tex_in = Tex("Fade", "In").scale(3)
            tex_out = Tex("Fade", "Out").scale(3)
            self.play(FadeIn(tex_in, shift=DOWN, scale=0.66))
            self.play(ReplacementTransform(tex_in, tex_out))
            self.play(FadeOut(tex_out, shift=DOWN * 2, scale=1.5))

    """

from __future__ import annotations

__all__ = [
    "FadeOut",
    "FadeIn",
]
import numpy as np

from manim.mobject.opengl.opengl_mobject import OpenGLMobject

from ..animation.transform import Transform
from ..constants import ORIGIN
from ..mobject.mobject import Group, Mobject
from ..scene.scene import Scene

class _Fade(Transform):
    """Fade :class:`~.Mobject`'s in or out.

    Parameters
    -----
    mobjects
        The mobjects to be faded.
    shift
        The vector by which the mobject shifts while being faded.
    target_position
        The position to/from which the mobject moves while being faded in. In case
        another mobject is given as target position, its center is used.
    scale
        The factor by which the mobject is scaled initially before being rescaling to
        its original size while being faded in.

    """
    def __init__(self,
                 *mobjects: Mobject,
                 shift: np.ndarray | None = None,
                 target_position: np.ndarray | Mobject | None = None,
                 scale: float = 1,
                 **kwargs,
                 ) -> None:
        if not mobjects:
            raise ValueError("At least one mobject must be passed.")
        if len(mobjects) == 1:
            mobject = mobjects[0]
        else:
            mobject = Group(*mobjects)

        self.point_target = False
        if shift is None:
            if target_position is not None:
                if isinstance(target_position, (Mobject, OpenGLMobject)):
                    target_position = target_position.get_center()
                shift = target_position - mobject.get_center()
                self.point_target = True
            else:
                shift = ORIGIN
        self.shift_vector = shift
        self.scale_factor = scale
        super().__init__(mobject, **kwargs)

    def _create_faded_mobject(self, fadeIn: bool) -> Mobject:
        """Create a faded, shifted and scaled copy of the mobject.
```

```

Parameters
-----
fadeIn
    Whether the faded mobject is used to fade in.

Returns
-----
Mobject
    The faded, shifted and scaled copy of the mobject.

"""
faded_mobject = self.mobject.copy()
faded_mobject.fade(1)
direction_modifier = -1 if fadeIn and not self.point_target else 1
faded_mobject.shift(self.shift_vector * direction_modifier)
faded_mobject.scale(self.scale_factor)
return faded_mobject

class FadeIn(_Fade):
    """Fade in :class:`~.Mobject` s.

Parameters
-----
mobjects
    The mobjects to be faded in.
shift
    The vector by which the mobject shifts while being faded in.
target_position
    The position from which the mobject starts while being faded in. In case
    another mobject is given as target position, its center is used.
scale
    The factor by which the mobject is scaled initially before being rescaling to
    its original size while being faded in.

Examples
-----
.. manim :: FadeInExample

class FadeInExample(Scene):
    def construct(self):
        dot = Dot(UP * 2 + LEFT)
        self.add(dot)
        tex = Tex(
            "FadeIn with ", "shift ", " or target\\_position", " and scale"
        ).scale(1)
        animations = [
            FadeIn(tex[0]),
            FadeIn(tex[1], shift=DOWN),
            FadeIn(tex[2], target_position=dot),
            FadeIn(tex[3], scale=1.5),
        ]
        self.play(AnimationGroup(*animations, lag_ratio=0.5))

"""

def __init__(self, *mobjects: Mobject, **kwargs) -> None:
    super().__init__(*mobjects, introducer=True, **kwargs)

def create_target(self):
    return self.mobject

def create_starting_mobject(self):
    return self._create_faded_mobject(fadeIn=True)

class FadeOut(_Fade):
    """Fade out :class:`~.Mobject` s.

Parameters
-----
mobjects
    The mobjects to be faded out.
shift
    The vector by which the mobject shifts while being faded out.
target_position
    The position to which the mobject moves while being faded out. In case another
    mobject is given as target position, its center is used.
scale
    The factor by which the mobject is scaled while being faded out.

Examples
-----
.. manim :: FadeInExample

class FadeInExample(Scene):

```

```
def construct(self):
    dot = Dot(UP * 2 + LEFT)
    self.add(dot)
    tex = Tex(
        "FadeOut with ", "shift ", " or target\\_position", " and scale"
    ).scale(1)
    animations = [
        FadeOut(tex[0]),
        FadeOut(tex[1], shift=DOWN),
        FadeOut(tex[2], target_position=dot),
        FadeOut(tex[3], scale=0.5),
    ]
    self.play(AnimationGroup(*animations, lag_ratio=0.5))

"""

def __init__(self, *mobjects: Mobject, **kwargs) -> None:
    super().__init__(*mobjects, remover=True, **kwargs)

def create_target(self):
    return self._create_faded_mobject(fadeIn=False)

def clean_up_from_scene(self, scene: Scene = None) -> None:
    super().clean_up_from_scene(scene)
    self.interpolate(0)
```

functions.py

```
"""Mobjects representing function graphs."""

from __future__ import annotations

__all__ = ["ParametricFunction", "FunctionGraph", "ImplicitFunction"]

from typing import Callable, Iterable, Sequence

import numpy as np
from isosurfaces import plot_isoline

from manim import config
from manim.mobject.graphing.scale import LinearBase, _ScaleBase
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.types.vectorized_mobject import VМобјект
from manim.utils.color import YELLOW

class ParametricFunction(VМобјект, metaclass=ConvertToOpenGL):
    """A parametric curve.

    Parameters
    -----
    function
        The function to be plotted in the form of ``(lambda x: x**2)``
    t_range
        Determines the length that the function spans. By default ``[0, 1]``
    scaling
        Scaling class applied to the points of the function. Default of :class:`~.LinearBase`.
    use_smoothing
        Whether to interpolate between the points of the function after they have been created.
        (Will have odd behaviour with a low number of points)
    use_vectorized
        Whether to pass in the generated t value array to the function as ``[t_0, t_1, ...]``.
        Only use this if your function supports it. Output should be a numpy array
        of shape ``[[x_0, x_1, ...], [y_0, y_1, ...], [z_0, z_1, ...]]`` but ``z`` can
        also be 0 if the Axes is 2D
    discontinuities
        Values of t at which the function experiences discontinuity.
    dt
        The left and right tolerance for the discontinuities.

    Examples
    -----
    .. manim:: PlotParametricFunction
       :save_last_frame:

       class PlotParametricFunction(Scene):
           def func(self, t):
               return np.array((np.sin(2 * t), np.sin(3 * t), 0))

           def construct(self):
               func = ParametricFunction(self.func, t_range = np.array([0, TAU]), fill_opacity=0).set_color(RED)
               self.add(func.scale(3))

    .. manim:: ThreeDParametricSpring
       :save_last_frame:

       class ThreeDParametricSpring(ThreeDScene):
           def construct(self):
               curvel = ParametricFunction(
                   lambda u: np.array([
                       1.2 * np.cos(u),
                       1.2 * np.sin(u),
                       u * 0.05
                   ]), color=RED, t_range = np.array([-3*TAU, 5*TAU, 0.01]))
               curvel.set_shade_in_3d(True)
               axes = ThreeDAxes()
               self.add(axes, curvel)
               self.set_camera_orientation(phi=80 * DEGREES, theta=-60 * DEGREES)
               self.wait()

    .. attention::
       If your function has discontinuities, you'll have to specify the location
       of the discontinuities manually. See the following example for guidance.

    .. manim:: DiscontinuousExample
       :save_last_frame:
```

```

class DiscontinuousExample(Scene):
    def construct(self):
        ax1 = NumberPlane((-3, 3), (-4, 4))
        ax2 = NumberPlane((-3, 3), (-4, 4))
        VGroup(ax1, ax2).arrange()
        discontinuous_function = lambda x: (x ** 2 - 2) / (x ** 2 - 4)
        incorrect = ax1.plot(discontinuous_function, color=RED)
        correct = ax2.plot(
            discontinuous_function,
            discontinuities=[-2, 2], # discontinuous points
            dt=0.1, # left and right tolerance of discontinuity
            color=GREEN,
        )
        self.add(ax1, ax2, incorrect, correct)
    """
    def __init__(self,
                 function: Callable[[float, float], float],
                 t_range: Sequence[float] | None = None,
                 scaling: _ScaleBase = LinearBase(),
                 dt: float = 1e-8,
                 discontinuities: Iterable[float] | None = None,
                 use_smoothing: bool = True,
                 use_vectorized: bool = False,
                 **kwargs,
                 ):
        self.function = function
        t_range = [0, 1, 0.01] if t_range is None else t_range
        if len(t_range) == 2:
            t_range = np.array([*t_range, 0.01])

        self.scaling = scaling

        self.dt = dt
        self.discontinuities = discontinuities
        self.use_smoothing = use_smoothing
        self.use_vectorized = use_vectorized
        self.t_min, self.t_max, self.t_step = t_range

        super().__init__(**kwargs)

    def get_function(self):
        return self.function

    def get_point_from_function(self, t):
        return self.function(t)

    def generate_points(self):
        if self.discontinuities is not None:
            discontinuities = filter(
                lambda t: self.t_min <= t <= self.t_max,
                self.discontinuities,
            )
            discontinuities = np.array(list(discontinuities))
            boundary_times = np.array([
                self.t_min,
                self.t_max,
                *(discontinuities - self.dt),
                *(discontinuities + self.dt),
            ],
            )
            boundary_times.sort()
        else:
            boundary_times = [self.t_min, self.t_max]

        for t1, t2 in zip(boundary_times[0::2], boundary_times[1::2]):
            t_range = np.array([
                *self.scaling.function(np.arange(t1, t2, self.t_step)),
                self.scaling.function(t2),
            ],
            )

            if self.use_vectorized:
                x, y, z = self.function(t_range)
                if not isinstance(z, np.ndarray):
                    z = np.zeros_like(x)
                points = np.stack([x, y, z], axis=1)
            else:
                points = np.array([self.function(t) for t in t_range])

            self.start_new_path(points[0])
            self.add_points_as_corners(points[1:])
        if self.use_smoothing:
            # TODO: not in line with upstream, approx_smooth does not exist

```

```

        self.make_smooth()
    return self

init_points = generate_points

class FunctionGraph(ParametricFunction):
    """A :class:`ParametricFunction` that spans the length of the scene by default.

Examples
-----
.. manim:: ExampleFunctionGraph
    :save_last_frame:

    class ExampleFunctionGraph(Scene):
        def construct(self):
            cos_func = FunctionGraph(
                lambda t: np.cos(t) + 0.5 * np.cos(7 * t) + (1 / 7) * np.cos(14 * t),
                color=RED,
            )

            sin_func_1 = FunctionGraph(
                lambda t: np.sin(t) + 0.5 * np.sin(7 * t) + (1 / 7) * np.sin(14 * t),
                color=BLUE,
            )

            sin_func_2 = FunctionGraph(
                lambda t: np.sin(t) + 0.5 * np.sin(7 * t) + (1 / 7) * np.sin(14 * t),
                x_range=[-4, 4],
                color=GREEN,
            ).move_to([0, 1, 0])

            self.add(cos_func, sin_func_1, sin_func_2)
"""

def __init__(self, function, x_range=None, color=YELLOW, **kwargs):
    if x_range is None:
        x_range = np.array([-config["frame_x_radius"], config["frame_x_radius"]])

    self.x_range = x_range
    self.parametric_function = lambda t: np.array([t, function(t), 0])
    self.function = function
    super().__init__(self.parametric_function, self.x_range, color=color, **kwargs)

def get_function(self):
    return self.function

def get_point_from_function(self, x):
    return self.parametric_function(x)

class ImplicitFunction(VMobject, metaclass=ConvertToOpenGL):
    def __init__(
        self,
        func: Callable[[float, float], float],
        x_range: Sequence[float] | None = None,
        y_range: Sequence[float] | None = None,
        min_depth: int = 5,
        max_quads: int = 1500,
        use_smoothing: bool = True,
        **kwargs,
    ):
        """An implicit function.

Parameters
-----
func
    The implicit function in the form ``f(x, y) = 0``.
x_range
    The x min and max of the function.
y_range
    The y min and max of the function.
min_depth
    The minimum depth of the function to calculate.
max_quads
    The maximum number of quads to use.
use_smoothing
    Whether or not to smoothen the curves.
kwargs
    Additional parameters to pass into :class:`VMobject`


.. note::
    A small ``min_depth`` :math:`d` means that some small details might
    be ignored if they don't cross an edge of one of the
    :math:`4^d` uniform quads.
"""

```

The value of ``max_quads`` strongly corresponds to the quality of the curve, but a higher number of quads may take longer to render.

```
Examples
-----
.. manim:: ImplicitFunctionExample
    :save_last_frame:

    class ImplicitFunctionExample(Scene):
        def construct(self):
            graph = ImplicitFunction(
                lambda x, y: x * y ** 2 - x ** 2 * y - 2,
                color=YELLOW
            )
            self.add(NumberPlane(), graph)

    """
    self.function = func
    self.min_depth = min_depth
    self.max_quads = max_quads
    self.use_smoothing = use_smoothing
    self.x_range = x_range or [
        -config.frame_width / 2,
        config.frame_width / 2,
    ]
    self.y_range = y_range or [
        -config.frame_height / 2,
        config.frame_height / 2,
    ]
    super().__init__(**kwargs)

def generate_points(self):
    p_min, p_max = (
        np.array([self.x_range[0], self.y_range[0]]),
        np.array([self.x_range[1], self.y_range[1]]),
    )
    curves = plot_isoline(
        fn=lambda u: self.function(u[0], u[1]),
        pmin=p_min,
        pmax=p_max,
        min_depth=self.min_depth,
        max_quads=self.max_quads,
    ) # returns a list of lists of 2D points
    curves = [
        np.pad(curve, [(0, 0), (0, 1)]) for curve in curves if curve != []
    ] # add z coord as 0
    for curve in curves:
        self.start_new_path(curve[0])
        self.add_points_as_corners(curve[1:])
    if self.use_smoothing:
        self.make_smooth()
    return self

init_points = generate_points
```

scene_file_writer.py

```
"""The interface between scenes and ffmpeg."""
from __future__ import annotations
__all__ = ["SceneFileWriter"]

import json
import os
import shutil
import subprocess
from pathlib import Path
from typing import TYPE_CHECKING, Any

import numpy as np
import srt
from PIL import Image
from pydub import AudioSegment

from manim import __version__

from .. import config, logger
from ..config.logger_utils import set_file_logger
from ..constants import Renderertype
from ..utils.file_ops import (
    add_extension_if_not_present,
    add_version_before_extension,
    ensure_executable,
    guarantee_existence,
    is_gif_format,
    is_png_format,
    is_webm_format,
    is_webm_format,
    modify_atime,
    write_to_movie,
)
from ..utils.sounds import get_full_sound_file_path
from .section import DefaultSectionType, Section

if TYPE_CHECKING:
    from manim.renderer.opengl_renderer import OpenGLRenderer

class SceneFileWriter:
    """
    SceneFileWriter is the object that actually writes the animations
    played, into video files, using FFMPEG.
    This is mostly for Manim's internal use. You will rarely, if ever,
    have to use the methods for this class, unless tinkering with the very
    fabric of Manim's reality.

    Attributes
    -----
    sections : list of :class:`.Section`
        used to segment scene
    sections_output_dir : :class:`pathlib.Path`
        where are section videos stored
    output_name : str
        name of movie without extension and basis for section video names

    Some useful attributes are:
    "write_to_movie" (bool=False)
        Whether or not to write the animations into a video file.
    "movie_file_extension" (str=".mp4")
        The file-type extension of the outputted video.
    "partial_movie_files"
        List of all the partial-movie files.

    """
    force_output_as_scene_name = False

    def __init__(self, renderer, scene_name, **kwargs):
        self.renderer = renderer
        self.init_output_directories(scene_name)
        self.init_audio()
        self.frame_count = 0
        self.partial_movie_files: list[str] = []
        self.subcaptions: list[srt.Subtitle] = []
        self.sections: list[Section] = []
        # first section gets automatically created for convenience
        # if you need the first section to be skipped, add a first section by hand, it will replace this one
        self.next_section(
            name="autocreated", type=DefaultSectionType.NORMAL, skip_animations=False
        )
        # fail fast if ffmpeg is not found
        if not ensure_executable(Path(config.ffmpeg_executable)):
            raise RuntimeError(
                "Manim could not find ffmpeg, which is required for generating video output.\n"
                "For installing ffmpeg please consult https://docs.manim.community/en/stable/installation.html\n"
                "Make sure to either add ffmpeg to the PATH environment variable\n"
                "or set path to the ffmpeg executable under the ffmpeg header in Manim's configuration."
            )

    def init_output_directories(self, scene_name):
        """
        Initialise output directories.

        Notes
        -----
        The directories are read from ``config``, for example
        ``config['media_dir']``. If the target directories don't already
        exist, they will be created.

        """
        if config["dry_run"]:
            # in dry-run mode there is no output
            return

        if config["input_file"]:
            module_name = config.get_dir("input_file").stem
        else:
            module_name = ""

        if SceneFileWriter.force_output_as_scene_name:
            self.output_name = Path(scene_name)
        elif config["output_file"] and not config["write_all"]:
            self.output_name = config.get_dir("output_file")
        else:
            self.output_name = Path(scene_name)

        if config["media_dir"]:
            image_dir = guarantee_existence(
                config.get_dir(
                    "images_dir", module_name=module_name, scene_name=scene_name
                ),
            )
            self.image_file_path = image_dir / add_extension_if_not_present(
                self.output_name, ".png"
            )
        
```

```

if write_to_movie():
    movie_dir = guarantee_existence(
        config.get_dir(
            "video_dir", module_name=module_name, scene_name=scene_name
        ),
    )
    self.movie_file_path = movie_dir / add_extension_if_not_present(
        self.output_name, config["movie_file_extension"]
    )

    # TODO: /dev/null would be good in case sections_output_dir is used without being set (doesn't work on Windows), everyone likes defensive programming, right?
    self.sections_output_dir = Path("")
    if config.save_sections:
        self.sections_output_dir = guarantee_existence(
            config.get_dir(
                "sections_dir", module_name=module_name, scene_name=scene_name
            )
        )

    if is_gif_format():
        self.gif_file_path = add_extension_if_not_present(
            self.output_name, ".gif"
        )

    if not config["output_file"]:
        self.gif_file_path = add_version_before_extension(
            self.gif_file_path
        )

    self.gif_file_path = movie_dir / self.gif_file_path

    self.partial_movie_directory = guarantee_existence(
        config.get_dir(
            "partial_movie_dir",
            scene_name=scene_name,
            module_name=module_name,
        ),
    )

    if config["log_to_file"]:
        log_dir = guarantee_existence(config.get_dir("log_dir"))
        set_file_logger(
            scene_name=scene_name, module_name=module_name, log_dir=log_dir
        )

def finish_last_section(self) -> None:
    """Delete current section if it is empty."""
    if len(self.sections) and self.sections[-1].is_empty():
        self.sections.pop()

def next_section(self, name: str, type: str, skip_animations: bool) -> None:
    """Create segmentation cut here."""
    self.finish_last_section()

    # images don't support sections
    section_video: str | None = None
    # don't save when None
    if (
        not config.dry_run
        and write_to_movie()
        and config.save_sections
        and not skip_animations
    ):
        # relative to index file
        section_video = f"{self.output_name}_{len(self.sections):04}{config.movie_file_extension}"

    self.sections.append(
        Section(
            type,
            section_video,
            name,
            skip_animations,
        ),
    )

def add_partial_movie_file(self, hash_animation: str):
    """Adds a new partial movie file path to `scene.partial_movie_files` and current section from a hash.
    This method will compute the path from the hash. In addition to that it adds the new animation to the current section.

    Parameters
    -----
    hash_animation
        Hash of the animation.
    """
    if not hasattr(self, "partial_movie_directory") or not write_to_movie():
        return

    # None has to be added to partial_movie_files to keep the right index with scene.num_plays.
    # i.e if an animation is skipped, scene.num_plays is still incremented and we add an element to partial_movie_file be even with num_plays.
    if hash_animation is None:
        self.partial_movie_files.append(None)
        self.sections[-1].partial_movie_files.append(None)
    else:
        new_partial_movie_file = str(
            self.partial_movie_directory
            / f"(hash_animation){config['movie_file_extension']}"
        )
        self.partial_movie_files.append(new_partial_movie_file)
        self.sections[-1].partial_movie_files.append(new_partial_movie_file)

def get_resolution_directory(self):
    """Get the name of the resolution directory directly containing the video file.

    This method gets the name of the directory that immediately contains the video file. This name is ``height_in_pixels_of_video>p<frame_rate>``.
    For example, if you are rendering an 854x480 px animation at 15fps, the name of the directory that immediately contains the video, file will be ``480p15``.

    The file structure should look something like::

        MEDIA_DIR
        |--Tex
        |--texts
        |--videos
        |--<name_of_file_containing_scene>
        |--<height_in_pixels_of_video>p<frame_rate>
        |--<scene_name>.mp4

    Returns
    -----
    :class:`str`
        The name of the directory.
    """
    pixel_height = config["pixel_height"]
    frame_rate = config["frame_rate"]
    return f"({pixel_height})p{frame_rate}"

# Sound
def init_audio(self):
    """
    Prep the writer for adding audio to the movie.

```

```

"""
self.includes_sound = False

def create_audio_segment(self):
    """
    Creates an empty, silent, Audio Segment.
    """
    self.audio_segment = AudioSegment.silent()

def add_audio_segment(
    self,
    new_segment: AudioSegment,
    time: float | None = None,
    gain_to_background: float | None = None,
):
    """
    This method adds an audio segment from an
    AudioSegment type object and suitable parameters.

    Parameters
    -----
    new_segment
        The audio segment to add

    time
        the timestamp at which the
        sound should be added.

    gain_to_background
        The gain of the segment from the background.
    """
    if not self.includes_sound:
        self.includes_sound = True
        self.create_audio_segment()
    segment = self.audio_segment
    curr_end = segment.duration_seconds
    if time is None:
        time = curr_end
    if time < 0:
        raise ValueError("Adding sound at timestamp < 0")

    new_end = time + new_segment.duration_seconds
    diff = new_end - curr_end
    if diff > 0:
        segment = segment.append(
            AudioSegment.silent(int(np.ceil(diff * 1000))),
            crossfade=0,
        )
    self.audio_segment = segment.overlay(
        new_segment,
        position=int(1000 * time),
        gain_during_overlay=gain_to_background,
    )

def add_sound(
    self,
    sound_file: str,
    time: float | None = None,
    gain: float | None = None,
    **kwargs,
):
    """
    This method adds an audio segment from a sound file.

    Parameters
    -----
    sound_file
        The path to the sound file.

    time
        The timestamp at which the audio should be added.

    gain
        The gain of the given audio segment.

    **kwargs
        This method uses add_audio_segment, so any keyword arguments
        used there can be referenced here.

    """
    file_path = get_full_sound_file_path(sound_file)
    new_segment = AudioSegment.from_file(file_path)
    if gain:
        new_segment = new_segment.apply_gain(gain)
    self.add_audio_segment(new_segment, time, **kwargs)

# Writers
def begin_animation(self, allow_write: bool = False, file_path=None):
    """
    Used internally by manim to stream the animation to FFMPEG for
    displaying or writing to a file.

    Parameters
    -----
    allow_write
        Whether or not to write to a video file.
    """
    if write_to_movie() and allow_write:
        self.open_movie_pipe(file_path=file_path)

def end_animation(self, allow_write: bool = False):
    """
    Internally used by Manim to stop streaming to
    FFMPEG gracefully.

    Parameters
    -----
    allow_write
        Whether or not to write to a video file.
    """
    if write_to_movie() and allow_write:
        self.close_movie_pipe()

def write_frame(self, frame_or_renderer: np.ndarray | OpenGLRenderer):
    """
    Used internally by Manim to write a frame to
    the FFMPEG input buffer.

    Parameters
    -----
    frame_or_renderer
        Pixel array of the frame.
    """
    if config.renderer == RendererType.OPENGL:
        self.write_opengl_frame(frame_or_renderer)
    elif config.renderer == RendererType.CAIRO:
        frame = frame_or_renderer
        if write_to_movie():
            self.writing_process.stdin.write(frame.tobytes())
        if is_png_format() and not config["dry_run"]:
            self.output_image_from_array(frame)

def write_opengl_frame(self, renderer: OpenGLRenderer):
    if write_to_movie():

```

```

        self.writing_process.stdin.write(
            renderer.get_raw_frame_buffer_object_data(),
        )
    elif is_png_format() and not config["dry_run"]:
        target_dir = self.image_file_path.parent / self.image_file_path.stem
        extension = self.image_file_path.suffix
        self.output_image(
            renderer.get_image(),
            target_dir,
            extension,
            config["zero_pad"],
        )
    )

def output_image_from_array(self, frame_data):
    target_dir = self.image_file_path.parent / self.image_file_path.stem
    extension = self.image_file_path.suffix
    self.output_image(
        Image.fromarray(frame_data),
        target_dir,
        extension,
        config["zero_pad"],
    )

def output_image(self, image: Image.Image, target_dir, ext, zero_pad: bool):
    if zero_pad:
        image.save(f"{target_dir}{str(self.frame_count).zfill(zero_pad)}{ext}")
    else:
        image.save(f"{target_dir}{self.frame_count}{ext}")
    self.frame_count += 1

def save_final_image(self, image: np.ndarray):
    """
    The name is a misnomer. This method saves the image
    passed to it as an in the default image directory.

    Parameters
    -----
    image
        The pixel array of the image to save.
    """
    if config["dry_run"]:
        return
    if not config["output_file"]:
        self.image_file_path = add_version_before_extension(self.image_file_path)
    image.save(self.image_file_path)
    self.print_file_ready_message(self.image_file_path)

def finish(self):
    """
    Finishes writing to the FFMPEG buffer or writing images
    to output directory.
    Combines the partial movie files into the
    whole scene.
    If save_last_frame is True, saves the last
    frame in the default image directory.
    """
    if write_to_movie():
        if hasattr(self, "writing_process"):
            self.writing_process.terminate()
        self.combine_to_movie()
        if config.save_sections:
            self.combine_to_section_videos()
        if config["flush_cache"]:
            self.flush_cache_directory()
        else:
            self.clean_cache()
    elif is_png_format() and not config["dry_run"]:
        target_dir = self.image_file_path.parent / self.image_file_path.stem
        logger.info("\n% images ready at %s\n", self.frame_count, str(target_dir))
    if self.subcaptions:
        self.write_subcaption_file()

def open_movie_pipe(self, file_path=None):
    """
    Used internally by Manim to initialise
    FFMPEG and begin writing to FFMPEG's input
    buffer.
    """
    if file_path is None:
        file_path = self.partial_movie_files[self.renderer.num_plays]
        self.partial_movie_file_path = file_path

    fps = config["frame_rate"]
    if fps == int(fps): # fps is integer
        fps = int(fps)
    if config.renderer == RendererType.OPENGL:
        width, height = self.renderer.get_pixel_shape()
    else:
        height = config["pixel_height"]
        width = config["pixel_width"]

    command = [
        config.ffmpeg_executable,
        "-y", # overwrite output file if it exists
        "-f",
        "rawvideo",
        "-s",
        f"-s{width}x{height} # size of one frame",
        "-pix_fmt",
        "rgba",
        "-r",
        str(fps), # frames per second
        "-i",
        "-", # The input comes from a pipe
        "-an", # Tells FFMPEG not to expect any audio
        "-loglevel",
        config["ffmpeg_loglevel"].lower(),
        "-metadata",
        "-comment=Rendered with Manim Community v{__version__}",
    ]
    if config.renderer == RendererType.OPENGL:
        command += ["-vf", "vflip"]
    if is_webm_format():
        command += ["-vcodec", "libvpx-vp9", "-auto-alt-ref", "0"]
    # .mov format
    elif config["transparent"]:
        command += ["-vcodec", "qtrle"]
    else:
        command += ["-vcodec", "libx264", "-pix_fmt", "yuv420p"]
    command += [file_path]
    self.writing_process = subprocess.Popen(command, stdin=subprocess.PIPE)

def close_movie_pipe(self):
    """
    Used internally by Manim to gracefully stop writing to FFMPEG's input buffer
    """
    self.writing_process.stdin.close()
    self.writing_process.wait()

    logger.info(
        f"Animation {self.renderer.num_plays} : Partial movie file written in %(path)s",
        {"path": f'{self.partial_movie_file_path}'},
    )
}

```

```

def is_already_cached(self, hash_invocation: str):
    """Will check if a file named with `hash_invocation` exists.

    Parameters
    -----
    hash_invocation
        The hash corresponding to an invocation to either `scene.play` or `scene.wait`.

    Returns
    -----
    :class:`bool`
        Whether the file exists.
    """
    if not hasattr(self, "partial_movie_directory") or not write_to_movie():
        return False
    path = (
        self.partial_movie_directory
        / f'{hash_invocation}{config["movie_file_extension"]}'
    )
    return path.exists()

def combine_files(
    self,
    input_files: list[str],
    output_file: Path,
    create_gif=False,
    includes_sound=False,
):
    file_list = self.partial_movie_directory / "partial_movie_file_list.txt"
    logger.debug(
        f"Partial movie files to combine ({len(input_files)} files): {(p)s}",
        ("p": input_files[:5]),
    )
    with file_list.open("w", encoding="utf-8") as fp:
        fp.write("# This file is used internally by FFMPEG.\n")
        for pf_path in input_files:
            pf_path = Path(pf_path).as_posix()
            fp.write(f"file '{pf_path}'\n")
    commands = [
        config.ffmpeg_executable,
        "-y", # overwrite output file if it exists
        "-f",
        "concat",
        "-safe",
        "0",
        "-i",
        str(file_list),
        "-loglevel",
        config.ffmpeg_loglevel.lower(),
        "-metadata",
        f"comment=Rendered with Manim Community v{__version__}",
        "-nostdin",
    ]
    if create_gif:
        commands += [
            "-vf",
            f"-fps={np.clip(config['frame_rate'], 1, 50)},split[s0][s1];[s0]palettegen=stats_mode=diff[p];[s1][p]paletteuse=dither=bayer:bayer_scale=5:diff_mode=rectangle",
        ]
    else:
        commands += ["-c", "copy"]
    if not includes_sound:
        commands += ["-an"]
    commands += [str(output_file)]
    combine_process = subprocess.Popen(commands)
    combine_process.wait()

def combine_to_movie(self):
    """Used internally by Manim to combine the separate
    partial movie files that make up a Scene into a single
    video file for that Scene.

    """
    partial_movie_files = [el for el in self.partial_movie_files if el is not None]
    # NOTE: Here we should do a check and raise an exception if partial
    # movie file is empty. We can't, as a lot of stuff (in particular, in
    # tests) use scene initialization, and this error would be raised as
    # it's just an empty scene initialized.

    # determine output path
    movie_file_path = self.movie_file_path
    if is_gif_format():
        movie_file_path = self.gif_file_path
    logger.info("Combining to Movie file.")
    self.combine_files(
        partial_movie_files,
        movie_file_path,
        is_gif_format(),
        self.includes_sound,
    )

    # handle sound
    if self.includes_sound:
        sound_file_path = movie_file_path.with_suffix(".wav")
        # Makes sure sound file length will match video file
        self.add_audio_segment(AudioSegment.silent(0))
        self.audio_segment.export(
            sound_file_path,
            bitrate="312k",
        )
        temp_file_path = movie_file_path.with_name(
            f"{movie_file_path.stem}_temp{movie_file_path.suffix}"
        )
        commands = [
            config.ffmpeg_executable,
            "-i",
            str(movie_file_path),
            "-i",
            str(sound_file_path),
            "-y", # overwrite output file if it exists
            "-c:v:",
            "copy",
            "-c:a:",
            "aac",
            "-b:a:",
            "320K",
            # select video stream from first file
            "-map",
            "0:v:0",
            # select audio stream from second file
            "-map",
            "1:a:0",
            "-loglevel",
            config.ffmpeg_loglevel.lower(),
            "-metadata",
            f"comment=Rendered with Manim Community v{__version__}",
            # "-shortest",
            str(temp_file_path),
        ]
        subprocess.call(commands)
        shutil.move(str(temp_file_path), str(movie_file_path))

```

```

sound_file_path.unlink()

self.print_file_ready_message(str(movie_file_path))
if write_to_movie():
    for file_path in partial_movie_files:
        # We have to modify the accessed time so if we have to clean the cache we remove the one used the longest.
        modify_atime(file_path)

def combine_to_section_videos(self) -> None:
    """Concatenate partial movie files for each section."""

    self.finish_last_section()
    sections_index: list[dict[str, Any]] = []
    for section in self.sections:
        # only if section does want to be saved
        if section.video is not None:
            logger.info(f"Combining partial files for section '{section.name}'")
            self.combine_files(
                section.get_clean_partial_movie_files(),
                self.sections_output_dir / section.video,
            )
            sections_index.append(section.get_dict(self.sections_output_dir))
    with (self.sections_output_dir / f"{self.output_name}.json").open("w") as file:
        json.dump(sections_index, file, indent=4)

def clean_cache(self):
    """Will clean the cache by removing the oldest partial_movie_files."""
    cached_partial_movies = [
        (self.partial_movie_directory / file_name)
        for file_name in self.partial_movie_directory.iterdir()
        if file_name != "partial_movie_file_list.txt"
    ]
    if len(cached_partial_movies) > config["max_files_cached"]:
        number_files_to_delete = (
            len(cached_partial_movies) - config["max_files_cached"]
        )
        oldest_files_to_delete = sorted(
            cached_partial_movies,
            key=lambda path: path.stat().st_atime,
        )[:number_files_to_delete]
        for file_to_delete in oldest_files_to_delete:
            file_to_delete.unlink()
        logger.info(
            f"The partial movie directory is full (> {config['max_files_cached']}) files). Therefore, manim has removed the {number_files_to_delete} oldest file(s)."
            " You can change this behaviour by changing max_files_cached in config."
        )

def flush_cache_directory(self):
    """Delete all the cached partial movie files"""
    cached_partial_movies = [
        self.partial_movie_directory / file_name
        for file_name in self.partial_movie_directory.iterdir()
        if file_name != "partial_movie_file_list.txt"
    ]
    for f in cached_partial_movies:
        f.unlink()
    logger.info(
        f"Cache flushed. {len(cached_partial_movies)} file(s) deleted in %(par_dir)s.",
        {"par_dir": self.partial_movie_directory},
    )

def write_subcaption_file(self):
    """Writes the subcaption file."""
    subcaption_file = Path(config.output_file).with_suffix(".srt")
    subcaption_file.write_text(srt.compose(self.subcaptions), encoding="utf-8")
    logger.info(f"Subcaption file has been written as {subcaption_file}")

def print_file_ready_message(self, file_path):
    """Prints the "File Ready" message to STDOUT."""
    config["output_file"] = file_path
    logger.info("\nFile ready at %(file_path)s\n", {"file_path": f'{file_path}'})

```

opengl_renderer.py

```
from __future__ import annotations

import itertools as it
import sys
import time
from typing import Any

if sys.version_info < (3, 8):
    from backports.cached_property import cached_property
else:
    from functools import cached_property

import moderngl
import numpy as np
from PIL import Image

from manim import config, logger
from manim.mobject.opengl_mobject import OpenGLMobject, OpenGLPoint
from manim.mobject.opengl_vectorized_mobject import OpenGLVMobject
from manim.utils.caching import handle_caching_play
from manim.utils.color import color_to_rgba
from manim.utils.exceptions import EndSceneEarlyException

from ..constants import *
from ..scene.scene_file_writer import SceneFileWriter
from ..utils import opengl
from ..utils.config_ops import _Data
from ..utils.simple_functions import clip
from ..utils.space_ops import (
    angle_of_vector,
    quaternion_from_angle_axis,
    quaternion_mult,
    rotation_matrix_transpose,
    rotation_matrix_transpose_from_quaternion,
)
from .shader import Mesh, Shader
from .vectorized_mobject_rendering import (
    render_opengl_vectorized_mobject_fill,
    render_opengl_vectorized_mobject_stroke,
)
)

class OpenGLCamera(OpenGLMobject):
    euler_angles = _Data()

    def __init__(
        self,
        frame_shape=None,
        center_point=None,
        # Theta, phi, gamma
        euler_angles=[0, 0, 0],
        focal_distance=2,
        light_source_position=[-10, 10, 10],
        orthographic=False,
        minimum_polar_angle=-PI / 2,
        maximum_polar_angle=PI / 2,
        model_matrix=None,
        **kwargs,
    ):
        self.use_z_index = True
        self.frame_rate = 60
        self.orthographic = orthographic
        self.minimum_polar_angle = minimum_polar_angle
        self.maximum_polar_angle = maximum_polar_angle
        if self.orthographic:
            self.projection_matrix = opengl.orthographic_projection_matrix()
            self.unformatted_projection_matrix = opengl.orthographic_projection_matrix(
                format=False,
            )
        else:
            self.projection_matrix = opengl.perspective_projection_matrix()
            self.unformatted_projection_matrix = opengl.perspective_projection_matrix(
                format=False,
            )

        if frame_shape is None:
            self.frame_shape = (config["frame_width"], config["frame_height"])
        else:
            self.frame_shape = frame_shape

        if center_point is None:
```

```

        self.center_point = ORIGIN
    else:
        self.center_point = center_point

    if model_matrix is None:
        model_matrix = opengl.translation_matrix(0, 0, 11)

    self.focal_distance = focal_distance

    if light_source_position is None:
        self.light_source_position = [-10, 10, 10]
    else:
        self.light_source_position = light_source_position
    self.light_source = OpenGLPoint(self.light_source_position)

    self.default_model_matrix = model_matrix
super().__init__(model_matrix=model_matrix, should_render=False, **kwargs)

if euler_angles is None:
    euler_angles = [0, 0, 0]
euler_angles = np.array(euler_angles, dtype=float)

self.euler_angles = euler_angles
self.refresh_rotation_matrix()

def get_position(self):
    return self.model_matrix[:, 3][:3]

def set_position(self, position):
    self.model_matrix[:, 3][:3] = position
    return self

@property
def formatted_view_matrix(self):
    return opengl.matrix_to_shader_input(np.linalg.inv(self.model_matrix))

@property
def unformatted_view_matrix(self):
    return np.linalg.inv(self.model_matrix)

def init_points(self):
    self.set_points([ORIGIN, LEFT, RIGHT, DOWN, UP])
    self.set_width(self.frame_shape[0], stretch=True)
    self.set_height(self.frame_shape[1], stretch=True)
    self.move_to(self.center_point)

def to_default_state(self):
    self.center()
    self.set_height(config["frame_height"])
    self.set_width(config["frame_width"])
    self.set_euler_angles(0, 0, 0)
    self.model_matrix = self.default_model_matrix
    return self

def refresh_rotation_matrix(self):
    # Rotate based on camera orientation
    theta, phi, gamma = self.euler_angles
    quat = quaternion_mult(
        quaternion_from_angle_axis(theta, OUT, axis_normalized=True),
        quaternion_from_angle_axis(phi, RIGHT, axis_normalized=True),
        quaternion_from_angle_axis(gamma, OUT, axis_normalized=True),
    )
    self.inverse_rotation_matrix = rotation_matrix_transpose_from_quaternion(quat)

def rotate(self, angle, axis=OUT, **kwargs):
    curr_rot_T = self.inverse_rotation_matrix
    added_rot_T = rotation_matrix_transpose(angle, axis)
    new_rot_T = np.dot(curr_rot_T, added_rot_T)
    Fz = new_rot_T[2]
    phi = np.arccos(Fz[2])
    theta = angle_of_vector(Fz[:2]) + PI / 2
    partial_rot_T = np.dot(
        rotation_matrix_transpose(phi, RIGHT),
        rotation_matrix_transpose(theta, OUT),
    )
    gamma = angle_of_vector(np.dot(partial_rot_T, new_rot_T.T)[:, 0])
    self.set_euler_angles(theta, phi, gamma)
    return self

def set_euler_angles(self, theta=None, phi=None, gamma=None):
    if theta is not None:
        self.euler_angles[0] = theta
    if phi is not None:
        self.euler_angles[1] = phi
    if gamma is not None:
        self.euler_angles[2] = gamma
    self.refresh_rotation_matrix()

```

```

    return self

def set_theta(self, theta):
    return self.set_euler_angles(theta=theta)

def set_phi(self, phi):
    return self.set_euler_angles(phi=phi)

def set_gamma(self, gamma):
    return self.set_euler_angles(gamma=gamma)

def increment_theta(self, dtheta):
    self.euler_angles[0] += dtheta
    self.refresh_rotation_matrix()
    return self

def increment_phi(self, dphi):
    phi = self.euler_angles[1]
    new_phi = clip(phi + dphi, -PI / 2, PI / 2)
    self.euler_angles[1] = new_phi
    self.refresh_rotation_matrix()
    return self

def increment_gamma(self, dgamma):
    self.euler_angles[2] += dgamma
    self.refresh_rotation_matrix()
    return self

def get_shape(self):
    return (self.get_width(), self.get_height())

def get_center(self):
    # Assumes first point is at the center
    return self.points[0]

def get_width(self):
    points = self.points
    return points[2, 0] - points[1, 0]

def get_height(self):
    points = self.points
    return points[4, 1] - points[3, 1]

def get_focal_distance(self):
    return self.focal_distance * self.get_height()

def interpolate(self, *args, **kwargs):
    super().interpolate(*args, **kwargs)
    self.refresh_rotation_matrix()

points_per_curve = 3

class OpenGLRenderer:
    def __init__(self, file_writer_class=SceneFileWriter, skip_animations=False):
        # Measured in pixel widths, used for vector graphics
        self.anti_alias_width = 1.5
        self._file_writer_class = file_writer_class

        self._original_skipping_status = skip_animations
        self.skip_animations = skip_animations
        self.animation_start_time = 0
        self.animation_elapsed_time = 0
        self.time = 0
        self.animations_hashes = []
        self.num_plays = 0

        self.camera = OpenGLCamera()
        self.pressed_keys = set()

        # Initialize texture map.
        self.path_to_texture_id = {}

        self.background_color = config["background_color"]

    def init_scene(self, scene):
        self.partial_movie_files = []
        self.file_writer: Any = self._file_writer_class(
            self,
            scene.__class__.__name__,
        )
        self.scene = scene
        self.background_color = config["background_color"]
        if not hasattr(self, "window"):
            if self.should_create_window():
                from .opengl_renderer_window import Window

```

```

        self.window = Window(self)
        self.context = self.window.ctx
        self.frame_buffer_object = self.context.detect_framebuffer()
    else:
        self.window = None
        try:
            self.context = moderngl.create_context(standalone=True)
        except Exception:
            self.context = moderngl.create_context(
                standalone=True,
                backend="egl",
            )
        self.frame_buffer_object = self.get_frame_buffer_object(self.context, 0)
        self.frame_buffer_object.use()
    self.context.enable(moderngl.BLEND)
    self.context.wireframe = config["enable_wireframe"]
    self.context.blend_func = (
        moderngl.SRC_ALPHA,
        moderngl.ONE_MINUS_SRC_ALPHA,
        moderngl.ONE,
        moderngl.ONE,
    )

def should_create_window(self):
    if config["force_window"]:
        logger.warning(
            "'--force_window' is enabled, this is intended for debugging purposes "
            "and may impact performance if used when outputting files",
        )
    return True
    return (
        config["preview"]
        and not config["save_last_frame"]
        and not config["format"]
        and not config["write_to_movie"]
        and not config["dry_run"]
    )

def get_pixel_shape(self):
    if hasattr(self, "frame_buffer_object"):
        return self.frame_buffer_object.viewport[2:4]
    else:
        return None

def refresh_perspective_uniforms(self, camera):
    pw, ph = self.get_pixel_shape()
    fw, fh = camera.get_shape()
    # TODO, this should probably be a mobject uniform, with
    # the camera taking care of the conversion factor
    anti_alias_width = self.anti_alias_width / (ph / fh)
    # Orient light
    rotation = camera.inverse_rotation_matrix
    light_pos = camera.light_source.get_location()
    light_pos = np.dot(rotation, light_pos)

    self.perspective_uniforms = {
        "frame_shape": camera.get_shape(),
        "anti_alias_width": anti_alias_width,
        "camera_center": tuple(camera.get_center()),
        "camera_rotation": tuple(np.array(rotation).T.flatten()),
        "light_source_position": tuple(light_pos),
        "focal_distance": camera.get_focal_distance(),
    }

def render_mobject(self, mobject):
    if isinstance(mobject, OpenGLMObject):
        if config["use_projection_fill_shaders"]:
            render_opengl_vectorized_mobject_fill(self, mobject)

        if config["use_projection_stroke_shaders"]:
            render_opengl_vectorized_mobject_stroke(self, mobject)

    shader_wrapper_list = mobject.get_shader_wrapper_list()

    # Convert ShaderWrappers to Meshes.
    for shader_wrapper in shader_wrapper_list:
        shader = Shader(self.context, shader_wrapper.shader_folder)

        # Set textures.
        for name, path in shader_wrapper.texture_paths.items():
            tid = self.get_texture_id(path)
            shader.shader_program[name].value = tid

        # Set uniforms.
        for name, value in it.chain(
            shader_wrapper.uniforms.items(),

```

```

        self.perspective_uniforms.items(),
    ):

        try:
            shader.set_uniform(name, value)
        except KeyError:
            pass

    try:
        shader.set_uniform(
            "u_view_matrix", self.scene.camera.formatted_view_matrix
        )
        shader.set_uniform(
            "u_projection_matrix",
            self.scene.camera.projection_matrix,
        )
    except KeyError:
        pass

    # Set depth test.
    if shader_wrapper.depth_test:
        self.context.enable(moderngl.DEPTH_TEST)
    else:
        self.context.disable(moderngl.DEPTH_TEST)

    # Render.
    mesh = Mesh(
        shader,
        shader_wrapper.vert_data,
        indices=shader_wrapper.vert_indices,
        use_depth_test=shader_wrapper.depth_test,
        primitive=mobject.render_primitive,
    )
    mesh.set_uniforms(self)
    mesh.render()

def get_texture_id(self, path):
    if repr(path) not in self.path_to_texture_id:
        tid = len(self.path_to_texture_id)
        texture = self.context.texture(
            size=path.size,
            components=len(path.getbands()),
            data=path.tobytes(),
        )
        texture.repeat_x = False
        texture.repeat_y = False
        texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
        texture.swizzle = "RRR1" if path.mode == "L" else "RGBA"
        texture.use(location=tid)
        self.path_to_texture_id[repr(path)] = tid

    return self.path_to_texture_id[repr(path)]

def update_skipping_status(self):
    """
    This method is used internally to check if the current
    animation needs to be skipped or not. It also checks if
    the number of animations that were played correspond to
    the number of animations that need to be played, and
    raises an EndSceneEarlyException if they don't correspond.
    """
    # there is always at least one section -> no out of bounds here
    if self.file_writer.sections[-1].skip_animations:
        self.skip_animations = True
    if (
        config["from_animation_number"]
        and self.num_plays < config["from_animation_number"]
    ):
        self.skip_animations = True
    if (
        config["upto_animation_number"]
        and self.num_plays > config["upto_animation_number"]
    ):
        self.skip_animations = True
        raise EndSceneEarlyException()

@handle_caching_play
def play(self, scene, *args, **kwargs):
    # TODO: Handle data locking / unlocking.
    self.animation_start_time = time.time()
    self.file_writer.begin_animation(not self.skip_animations)

    scene.compile_animation_data(*args, **kwargs)
    scene.begin_animations()
    if scene.is_current_animation_frozen_frame():
        self.update_frame(scene)

        if not self.skip_animations:
            for _ in range(int(config.frame_rate * scene.duration)):
```

```

        self.file_writer.write_frame(self)

    if self.window is not None:
        self.window.swap_buffers()
        while time.time() - self.animation_start_time < scene.duration:
            pass
        self.animation_elapsed_time = scene.duration

    else:
        scene.play_internal()

    self.file_writer.end_animation(not self.skip_animations)
    self.time += scene.duration
    self.num_plays += 1

def clear_screen(self):
    self.frame_buffer_object.clear(*self.background_color)
    self.window.swap_buffers()

def render(self, scene, frame_offset, moving_mobjects):
    self.update_frame(scene)

    if self.skip_animations:
        return

    self.file_writer.write_frame(self)

    if self.window is not None:
        self.window.swap_buffers()
        while self.animation_elapsed_time < frame_offset:
            self.update_frame(scene)
            self.window.swap_buffers()

def update_frame(self, scene):
    self.frame_buffer_object.clear(*self.background_color)
    self.refresh_perspective_uniforms(scene.camera)

    for mobject in scene.mobjects:
        if not mobject.should_render:
            continue
        self.render_mobject(mobject)

    for obj in scene.meshes:
        for mesh in obj.get_meshes():
            mesh.set_uniforms(self)
            mesh.render()

    self.animation_elapsed_time = time.time() - self.animation_start_time

def scene_finished(self, scene):
    # When num_plays is 0, no images have been output, so output a single
    # image in this case
    if self.num_plays > 0:
        self.file_writer.finish()
    elif self.num_plays == 0 and config.write_to_movie:
        config.write_to_movie = False

    if self.should_save_last_frame():
        config.save_last_frame = True
        self.update_frame(scene)
        self.file_writer.save_final_image(self.get_image())

def should_save_last_frame(self):
    if config["save_last_frame"]:
        return True
    if self.scene.interactive_mode:
        return False
    return self.num_plays == 0

def get_image(self) -> Image.Image:
    """Returns an image from the current frame. The first argument passed to image represents
    the mode RGB with the alpha channel A. The data we read is from the currently bound frame
    buffer. We pass in 'raw' as the name of the decoder, 0 and -1 args are specifically
    used for the decoder and represent the stride and orientation. 0 means there is no
    padding expected between bytes and -1 represents the orientation and means the first
    line of the image is the bottom line on the screen.

    Returns
    ------
    PIL.Image
        The PIL image of the array.
    """
    raw_buffer_data = self.get_raw_frame_buffer_object_data()
    image = Image.frombytes(
        "RGBA",
        self.get_pixel_shape(),
        raw_buffer_data,

```

```

        "raw",
        "RGBA",
        0,
        -1,
    )
    return image

def save_static_frame_data(self, scene, static_mobjects):
    pass

def get_frame_buffer_object(self, context, samples=0):
    pixel_width = config["pixel_width"]
    pixel_height = config["pixel_height"]
    num_channels = 4
    return context.framebuffer(
        color_attachments=context.texture(
            (pixel_width, pixel_height),
            components=num_channels,
            samples=samples,
        ),
        depth_attachment=context.depth_renderbuffer(
            (pixel_width, pixel_height),
            samples=samples,
        ),
    )

def get_raw_frame_buffer_object_data(self, dtype="f1"):
    # Copy blocks from the fbo_msaa to the drawn fbo using Blit
    # pw, ph = self.get_pixel_shape()
    # gl.glBindFramebuffer(gl.GL_READ_FRAMEBUFFER, self.fbo_msaa.glo)
    # gl.glBindFramebuffer(gl.GL_DRAW_FRAMEBUFFER, self.fbo.glo)
    # gl.glBlitFramebuffer(
    #     0, 0, pw, ph, 0, 0, pw, ph, gl.GL_COLOR_BUFFER_BIT, gl.GL_LINEAR
    # )
    num_channels = 4
    ret = self.frame_buffer_object.read(
        viewport=self.frame_buffer_object.viewport,
        components=num_channels,
        dtype=dtype,
    )
    return ret

def get_frame(self):
    # get current pixel values as numpy data in order to test output
    raw = self.get_raw_frame_buffer_object_data(dtype="f1")
    pixel_shape = self.get_pixel_shape()
    result_dimensions = (pixel_shape[1], pixel_shape[0], 4)
    np_buf = np.frombuffer(raw, dtype="uint8").reshape(result_dimensions)
    np_buf = np.fliplr(np_buf)
    return np_buf

# Returns offset from the bottom left corner in pixels.
# top_left flag should be set to True when using a GUI framework
# where the (0,0) is at the top left: e.g. PySide6
def pixel_coords_to_space_coords(self, px, py, relative=False, top_left=False):
    pixel_shape = self.get_pixel_shape()
    if pixel_shape is None:
        return np.array([0, 0, 0])
    pw, ph = pixel_shape
    fw, fh = config["frame_width"], config["frame_height"]
    fc = self.camera.get_center()
    if relative:
        return 2 * np.array([px / pw, py / ph, 0])
    else:
        # Only scale wrt one axis
        scale = fh / ph
        return fc + scale * np.array([
            [(px - pw / 2), (-1 if top_left else 1) * (py - ph / 2), 0]
        ])

@property
def background_color(self):
    return self._background_color

@background_color.setter
def background_color(self, value):
    self._background_color = color_to_rgba(value, 1.0)

```

logo.py

```
"""Utilities for Manim's logo and banner."""

from __future__ import annotations
__all__ = ["ManimBanner"]

from manim.animation.updaters.update import UpdateFromAlphaFunc
from manim.mobject.geometry.arc import Circle
from manim.mobject.geometry.polygram import Square, Triangle
from manim.mobject.text.tex_mobject import MathTex, Tex

from ..animation.animation import override_animation
from ..animation.composition import AnimationGroup, Succession
from ..animation.creation import Create, SpiralIn
from ..animation.fading import FadeIn
from ..constants import DOWN, LEFT, ORIGIN, RIGHT, TAU, UP
from ..mobject.types.vectorized_mobject import VGroup
from ..utils.rate_functions import ease_in_out_cubic, ease_out_sine, smooth
from ..utils.tex_templates import TexFontTemplates


class ManimBanner(VGroup):
    r"""Convenience class representing Manim's banner.

    Can be animated using custom methods.

    Parameters
    -----
    dark_theme
        If ``True`` (the default), the dark theme version of the logo
        (with light text font) will be rendered. Otherwise, if ``False``,
        the light theme version (with dark text font) is used.

    Examples
    -----
    .. manim:: DarkThemeBanner

        class DarkThemeBanner(Scene):
            def construct(self):
                banner = ManimBanner()
                self.play(banner.create())
                self.play(banner.expand())
                self.wait()
                self.play(Unwrite(banner))

    .. manim:: LightThemeBanner

        class LightThemeBanner(Scene):
            def construct(self):
                self.camera.background_color = "#ece6e2"
                banner = ManimBanner(dark_theme=False)
                self.play(banner.create())
                self.play(banner.expand())
                self.wait()
                self.play(Unwrite(banner))

    """

    def __init__(self, dark_theme: bool = True):
        super().__init__()

        logo_green = "#81b29a"
        logo_blue = "#454866"
        logo_red = "#e07a5f"
        m_height_over_anim_height = 0.75748

        self.font_color = "#ece6e2" if dark_theme else "#343434"
        self.scale_factor = 1

        self.M = MathTex(r"\mathbb{M}").scale(7).set_color(self.font_color)
        self.M.shift(2.25 * LEFT + 1.5 * UP)

        self.circle = Circle(color=logo_green, fill_opacity=1).shift(LEFT)
        self.square = Square(color=logo_blue, fill_opacity=1).shift(UP)
        self.triangle = Triangle(color=logo_red, fill_opacity=1).shift(RIGHT)
        self.shapes = VGroup(self.triangle, self.square, self.circle)
        self.add(self.shapes, self.M)
        self.move_to(ORIGIN)

        anim = VGroup()
        for i, ch in enumerate("anim"):
```

```

tex = Tex(
    "\textbf{" + ch + "}",
    tex_template=TexFontTemplates.gnu_freeserif_freesans,
)
if i != 0:
    tex.next_to(anim, buff=0.01)
tex.align_to(self.M, DOWN)
anim.add(tex)
anim.set_color(self.font_color)
anim.height = m_height_over_anim_height * self.M.height

# Note: "anim" is only shown in the expanded state
# and thus not yet added to the submobjects of self.
self.anim = anim

def scale(self, scale_factor: float, **kwargs) -> ManimBanner:
    """Scale the banner by the specified scale factor.

    Parameters
    -----
    scale_factor
        The factor used for scaling the banner.

    Returns
    -----
    :class:`~.ManimBanner`
        The scaled banner.
    """
    self.scale_factor *= scale_factor
    # Note: self.anim is only added to self after expand()
    if self.anim not in self.submobjects:
        self.anim.scale(scale_factor, **kwargs)
    return super().scale(scale_factor, **kwargs)

@override_animation(Create)
def create(self, run_time: float = 2) -> AnimationGroup:
    """The creation animation for Manim's logo.

    Parameters
    -----
    run_time
        The run time of the animation.

    Returns
    -----
    :class:`~.AnimationGroup`
        An animation to be used in a :meth:`.Scene.play` call.
    """
    return AnimationGroup(
        SpiralIn(self.shapes, run_time=run_time),
        FadeIn(self.M, run_time=run_time / 2),
        lag_ratio=0.1,
    )

def expand(self, run_time: float = 1.5, direction="center") -> Succession:
    """An animation that expands Manim's logo into its banner.

    The returned animation transforms the banner from its initial
    state (representing Manim's logo with just the icons) to its
    expanded state (showing the full name together with the icons).

    See the class documentation for how to use this.

    .. note::

        Before calling this method, the text "anim" is not a
        submobject of the banner object. After the expansion,
        it is added as a submobject so subsequent animations
        to the banner object apply to the text "anim" as well.

    Parameters
    -----
    run_time
        The run time of the animation.
    direction
        The direction in which the logo is expanded.

    Returns
    -----
    :class:`~.Succession`
        An animation to be used in a :meth:`.Scene.play` call.

    Examples
    -----
    .. manim:: ExpandDirections

        class ExpandDirections(Scene):

```

```

    def construct(self):
        banners = [ManimBanner().scale(0.5).shift(UP*x) for x in [-2, 0, 2]]
        self.play(
            banners[0].expand(direction="right"),
            banners[1].expand(direction="center"),
            banners[2].expand(direction="left"),
        )

    """
    if direction not in ["left", "right", "center"]:
        raise ValueError("direction must be 'left', 'right' or 'center'.")"""

    m_shape_offset = 6.25 * self.scale_factor
    shape_sliding_overshoot = self.scale_factor * 0.8
    m_anim_buff = 0.06
    self.anim.next_to(self.M, buff=m_anim_buff).align_to(self.M, DOWN)
    self.anim.set_opacity(0)
    self.shapes.save_state()
    m_clone = self.anim[-1].copy()
    self.add(m_clone)
    m_clone.move_to(self.shapes)

    self.M.save_state()
    left_group = VGroup(self.M, self.anim, m_clone)

    def shift(vector):
        self.shapes.restore()
        left_group.align_to(self.M.saved_state, LEFT)
        if direction == "right":
            self.shapes.shift(vector)
        elif direction == "center":
            self.shapes.shift(vector / 2)
            left_group.shift(-vector / 2)
        elif direction == "left":
            left_group.shift(-vector)

    def slide_and_uncover(mob, alpha):
        shift(alpha * (m_shape_offset + shape_sliding_overshoot) * RIGHT)

        # Add letters when they are covered
        for letter in mob.anim:
            if mob.square.get_center()[0] > letter.get_center()[0]:
                letter.set_opacity(1)
                self.add(letter)

        # Finish animation
        if alpha == 1:
            self.remove(*[self.anim])
            self.add_to_back(self.anim)
            mob.shapes.set_z_index(0)
            mob.shapes.save_state()
            mob.M.save_state()

    def slide_back(mob, alpha):
        if alpha == 0:
            m_clone.set_opacity(1)
            m_clone.move_to(mob.anim[-1])
            mob.anim.set_opacity(1)

        shift(alpha * shape_sliding_overshoot * LEFT)

        if alpha == 1:
            mob.remove(m_clone)
            mob.add_to_back(mob.shapes)

    return Succession(
        UpdateFromAlphaFunc(
            self,
            slide_and_uncover,
            run_time=run_time * 2 / 3,
            rate_func=ease_in_out_cubic,
        ),
        UpdateFromAlphaFunc(
            self,
            slide_back,
            run_time=run_time * 1 / 3,
            rate_func=smooth,
        ),
    )
)

```

deprecation.py

```
"""Decorators for deprecating classes, functions and function parameters."""
from __future__ import annotations
__all__ = ["deprecated", "deprecated_params"]

import inspect
import re
from typing import Any, Callable, Iterable

from decorator import decorate, decorator

from .. import logger

def _get_callable_info(callable: Callable) -> tuple[str, str]:
    """Returns type and name of a callable.

    Parameters
    -----
    callable
        The callable

    Returns
    -----
    Tuple[str, str]
        The type and name of the callable. Type can be one of "class", "method" (for
        functions defined in classes) or "function". For methods, name is Class.method.
    """
    what = type(callable).__name__
    name = callable.__qualname__
    if what == "function" and "." in name:
        what = "method"
    elif what != "function":
        what = "class"
    return (what, name)

def _deprecation_text_component(
    since: str | None,
    until: str | None,
    message: str,
) -> str:
    """Generates a text component used in deprecation messages.

    Parameters
    -----
    since
        The version or date since deprecation
    until
        The version or date until removal of the deprecated callable
    message
        The reason for why the callable has been deprecated

    Returns
    -----
    str
        The deprecation message text component.
    """
    since = f"since {since}" if since else ""
    until = (
        f"is expected to be removed after {until}"
        if until
        else "may be removed in a later version"
    )
    msg = " " + message if message else ""
    return f"deprecated {since}and {until}.{msg}"

def deprecated(
    func: Callable = None,
    since: str | None = None,
    until: str | None = None,
    replacement: str | None = None,
    message: str | None = "",
) -> Callable:
    """Decorator to mark a callable as deprecated.

    The decorated callable will cause a warning when used. The docstring of the
    deprecated callable is adjusted to indicate that this callable is deprecated.

    Parameters
    -----
    func
        The function to be decorated. Should not be set by the user.
    since
        The version or date since deprecation.
    until
        The version or date until removal of the deprecated callable.
    replacement
        The identifier of the callable replacing the deprecated one.
    message
        The reason for why the callable has been deprecated.

    Returns
    -----
    Callable
        The decorated callable.

    Examples
    -----
    Basic usage::

        from manim.utils.deprecation import deprecated

        @deprecated
        def foo(**kwargs):
            pass
    
```

```

@deprecated
class Bar:
    def __init__(self):
        pass

    @deprecated
    def baz(self):
        pass

foo()
# WARNING The function foo has been deprecated and may be removed in a later version.

a = Bar()
# WARNING The class Bar has been deprecated and may be removed in a later version.

a.baz()
# WARNING The method Bar.baz has been deprecated and may be removed in a later version.

You can specify additional information for a more precise warning::

from manim.utils.deprecation import deprecated

@deprecated(
    since="v0.2",
    until="v0.4",
    replacement="bar",
    message="It is cooler."
)
def foo():
    pass

foo()
# WARNING The function foo has been deprecated since v0.2 and is expected to be removed after v0.4. Use bar instead. It is cooler.

You may also use dates instead of versions::

from manim.utils.deprecation import deprecated

@deprecated(since="05/01/2021", until="06/01/2021")
def foo():
    pass

foo()
# WARNING The function foo has been deprecated since 05/01/2021 and is expected to be removed after 06/01/2021.

"""
# If used as factory:
if func is None:
    return lambda func: deprecated(func, since, until, replacement, message)

what, name = _get_callable_info(func)

def warning_msg(for_docs: bool = False) -> str:
    """Generate the deprecation warning message.

    Parameters
    -----
    for_docs
        Whether or not to format the message for use in documentation.

    Returns
    -----
    str
        The deprecation message.
    """
    msg = message
    if replacement is not None:
        repl = replacement
        if for_docs:
            mapper = {"class": "class", "method": "meth", "function": "func"}
            repl = f":{mapper[what]}:{~.(replacement)}"
        msg = f"Use {repl} instead.{'' + message if message else ''}"
    deprecated = _deprecation_text_component(since, until, msg)
    return f"The {what} {name} has been {deprecated}"

def deprecate_docs(func: Callable):
    """Adjust docstring to indicate the deprecation.

    Parameters
    -----
    func
        The callable whose docstring to adjust.
    """
    warning = warning_msg(True)
    doc_string = func.__doc__ or ""
    func.__doc__ = f"{doc_string}\n.. attention:: Deprecated\n  {warning}"

def deprecate(func: Callable, *args, **kwargs):
    """The actual decorator used to extend the callables behavior.

    Logs a warning message.

    Parameters
    -----
    func
        The callable to decorate.
    args
        The arguments passed to the given callable.
    kwargs
        The keyword arguments passed to the given callable.

    Returns
    -----
    Any
        The return value of the given callable when being passed the given arguments.
    """
    logger.warning(warning_msg())
    return func(*args, **kwargs)

if type(func).__name__ != "function":
    deprecate_docs(func)
    func.__init__ = decorate(func.__init__, deprecate)
    return func

func = decorate(func, deprecate)

```

```

deprecate_docs(func)
return func

def deprecated_params(
    params: str | Iterable[str] | None = None,
    since: str | None = None,
    until: str | None = None,
    message: str | None = "",
    redirections: None
    | (Iterable[tuple[str, str] | Callable[..., dict[str, Any]]]) = None,
) -> Callable:
    """Decorator to mark parameters of a callable as deprecated.

    It can also be used to automatically redirect deprecated parameter values to their
    replacements.

    Parameters
    -----
    params
        The parameters to be deprecated. Can consist of:
        * An iterable of strings, with each element representing a parameter to deprecate
        * A single string, with parameter names separated by commas or spaces.
    since
        The version or date since deprecation.
    until
        The version or date until removal of the deprecated callable.
    message
        The reason for why the callable has been deprecated.
    redirections
        A list of parameter redirections. Each redirection can be one of the following:
        * A tuple of two strings. The first string defines the name of the deprecated
          parameter; the second string defines the name of the parameter to redirect to,
          when attempting to use the first string.
        * A function performing the mapping operation. The parameter names of the
          function determine which parameters are used as input. The function must
          return a dictionary which contains the redirected arguments.

    Redirected parameters are also implicitly deprecated.

    Returns
    -----
    Callable
        The decorated callable.

    Raises
    -----
    ValueError
        If no parameters are defined (neither explicitly nor implicitly).
    ValueError
        If defined parameters are invalid python identifiers.

    Examples
    -----
    Basic usage::

        from manim.utils.deprecation import deprecated_params

        @deprecated_params(params="a, b, c")
        def foo(**kwargs):
            pass

        foo(x=2, y=3, z=4)
        # No warning

        foo(a=2, b=3, z=4)
        # WARNING  The parameters a and b of method foo have been deprecated and may be removed in a later version.

    You can also specify additional information for a more precise warning::

        from manim.utils.deprecation import deprecated_params

        @deprecated_params(
            params="a, b, c",
            since="v0.2",
            until="v0.4",
            message="The letters x, y, z are cooler."
        )
        def foo(**kwargs):
            pass

        foo(a=2)
        # WARNING  The parameter a of method foo has been deprecated since v0.2 and is expected to be removed after v0.4. The letters x, y, z are cooler.

    Basic parameter redirection::

        from manim.utils.deprecation import deprecated_params

        @deprecated_params(redirections=[
            # Two ways to redirect one parameter to another:
            ("old_param", "new_param"),
            lambda old_param2: {"new_param2": old_param2}
        ])
        def foo(**kwargs):
            return kwargs

        foo(x=1, old_param=2)
        # WARNING  The parameter old_param of method foo has been deprecated and may be removed in a later version.
        # returns {"x": 1, "new_param": 2}

    Redirecting using a calculated value::

        from manim.utils.deprecation import deprecated_params

        @deprecated_params(redirections=[
            lambda runtime_in_ms: {"run_time": runtime_in_ms / 1000}
        ])
        def foo(**kwargs):
            return kwargs

        foo(runtime_in_ms=500)
        # WARNING  The parameter runtime_in_ms of method foo has been deprecated and may be removed in a later version.
        # returns {"run_time": 0.5}

```

```

Redirecting multiple parameter values to one::

from manim.utils.deprecation import deprecated_params

@deprecated_params(redirections=[
    lambda buff_x=1, buff_y=1: {"buff": (buff_x, buff_y)}
])
def foo(**kwargs):
    return kwargs

foo(buff_x=2)
# WARNING The parameter buff_x of method foo has been deprecated and may be removed in a later version.
# returns {"buff": (2, 1)}

Redirect one parameter to multiple::

from manim.utils.deprecation import deprecated_params

@deprecated_params(redirections=[
    lambda buff=1: {"buff_x": buff[0], "buff_y": buff[1]} if isinstance(buff, tuple)
    else {"buff_x": buff, "buff_y": buff}
])
def foo(**kwargs):
    return kwargs

foo(buff=0)
# WARNING The parameter buff of method foo has been deprecated and may be removed in a later version.
# returns {"buff_x": 0, buff_y: 0}

foo(buff=(1,2))
# WARNING The parameter buff of method foo has been deprecated and may be removed in a later version.
# returns {"buff_x": 1, buff_y: 2}

"""

# Check if decorator is used without parenthesis
if callable(params):
    raise ValueError("deprecate_params requires arguments to be specified.")

if params is None:
    params = []

# Construct params list
params = re.split(r"[,\s]+", params) if isinstance(params, str) else list(params)

# Add params which are only implicitly given via redirections
if redirections is None:
    redirections = []
for redirector in redirections:
    if isinstance(redirector, tuple):
        params.append(redirector[0])
    else:
        params.extend(list(inspect.signature(redirector).parameters))
# Keep ordering of params so that warning message is consistently the same
# This will also help pass unit testing
params = list(dict.fromkeys(params))

# Make sure params only contains valid identifiers
identifier = re.compile(r"^\w*\Z", re.UNICODE)
if not all(re.match(identifier, param) for param in params):
    raise ValueError("Given parameter values are invalid.")

redirections = list(redirections)

def warning_msg(func: Callable, used: list[str]):
    """Generate the deprecation warning message.

Parameters
-----
func
    The callable with deprecated parameters.
used
    The list of deprecated parameters used in a call.

Returns
-----
str
    The deprecation message.
"""
    what, name = _get_callable_info(func)
    plural = len(used) > 1
    parameter_s = "s" if plural else ""
    used_ = ", ".join(used[:-1]) + " and " + used[-1] if plural else used[0]
    has_have_been = "have been" if plural else "has been"
    deprecated = _deprecation_text_component(since, until, message)
    return f"The parameter{parameter_s} {used_} of {what} {name} {has_have_been} {deprecated}"

def redirect_params(kwargs: dict, used: list[str]):
    """Adjust the keyword arguments as defined by the redirections.

Parameters
-----
kwargs
    The keyword argument dictionary to be updated.
used
    The list of deprecated parameters used in a call.
"""
    for redirector in redirections:
        if isinstance(redirector, tuple):
            old_param, new_param = redirector
            if old_param in used:
                kwargs[new_param] = kwargs.pop(old_param)
        else:
            redirector_params = list(inspect.signature(redirector).parameters)
            redirector_args = {}
            for redirector_param in redirector_params:
                if redirector_param in used:
                    redirector_args[redirector_param] = kwargs.pop(redirector_param)
            if len(redirector_args) > 0:
                kwargs.update(redirector(**redirector_args))

def deprecate_params(func, *args, **kwargs):
    """The actual decorator function used to extend the callables behavior.

Logs a warning message when a deprecated parameter is used and redirects it if

```

```
specified.

Parameters
-----
func
    The callable to decorate.
args
    The arguments passed to the given callable.
kwargs
    The keyword arguments passed to the given callable.

Returns
-----
Any
    The return value of the given callable when being passed the given
    arguments.

"""
used = []
for param in params:
    if param in kwargs:
        used.append(param)

if len(used) > 0:
    logger.warning(warning_msg(func, used))
    redirect_params(kwargs, used)
return func(*args, **kwargs)

return decorator(deprecate_params)
```

point_cloud_mobject.py

```
"""Mobjects representing point clouds."""
from __future__ import annotations

__all__ = ["PMobject", "Mobject1D", "Mobject2D", "PGroup", "PointCloudDot", "Point"]

import numpy as np
from colour import Color

from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl_point_cloud import OpenGLPMobject

from ...constants import *
from ...mobject.mobject import Mobject
from ...utils.bezier import interpolate
from ...utils.color import (
    BLACK,
    WHITE,
    YELLOW,
    color_gradient,
    color_to_rgba,
    rgba_to_color,
)
from ...utils.iterables import stretch_array_to_length

class PMobject(Mobject, metaclass=ConvertToOpenGL):
    """A disc made of a cloud of Dots

    Examples
    -----
    .. manim:: PMobjectExample
        :save_last_frame:

    class PMobjectExample(Scene):
        def construct(self):
            pG = PGroup() # This is just a collection of PMobject's

            # As the scale factor increases, the number of points
            # removed increases.
            for sf in range(1, 9 + 1):
                p = PointCloudDot(density=20, radius=1).thin_out(sf)
                # PointCloudDot is a type of PMobject
                # and can therefore be added to a PGroup
                pG.add(p)

            # This organizes all the shapes in a grid.
            pG.arrange_in_grid()

            self.add(pG)

    """

    def __init__(self, stroke_width=DEFAULT_STROKE_WIDTH, **kwargs):
        self.stroke_width = stroke_width
        super().__init__(**kwargs)

    def reset_points(self):
        self.rgbas = np.zeros((0, 4))
        self.points = np.zeros((0, 3))
        return self

    def get_array_attrs(self):
        return super().get_array_attrs() + ["rgbas"]

    def add_points(self, points, rgbas=None, color=None, alpha=1):
        """Add points.

        Points must be a Nx3 numpy array.
        Rgbas must be a Nx4 numpy array if it is not None.
        """
        if not isinstance(points, np.ndarray):
            points = np.array(points)
        num_new_points = len(points)
        self.points = np.append(self.points, points, axis=0)
        if rgbas is None:
            color = Color(color) if color else self.color
            rgbas = np.repeat([color_to_rgba(color, alpha)], num_new_points, axis=0)
        elif len(rgbas) != len(points):
            raise ValueError("points and rgbas must have same length")
        self.rgbas = np.append(self.rgbas, rgbas, axis=0)
        return self

    def set_color(self, color=YELLOW, family=True):
        rgba = color_to_rgba(color)
        mobs = self.family_members_with_points() if family else [self]
        for mob in mobs:
            mob.rgbas[:, :] = rgba
        self.color = color
        return self

    def get_stroke_width(self):
        return self.stroke_width

    def set_stroke_width(self, width, family=True):
        mobs = self.family_members_with_points() if family else [self]
        for mob in mobs:
            mob.stroke_width = width
        return self

    def set_color_by_gradient(self, *colors):
        self.rgbas = np.array(
            list(map(color_to_rgba, color_gradient(*colors, len(self.points)))),
        )
        return self
```

```

def set_colors_by_radial_gradient(
    self,
    center=None,
    radius=1,
    inner_color=WHITE,
    outer_color=BLACK,
):
    start_rgba, end_rgba = list(map(color_to_rgba, [inner_color, outer_color]))
    if center is None:
        center = self.get_center()
    for mob in self.family_members_with_points():
        distances = np.abs(self.points - center)
        alphas = np.linalg.norm(distances, axis=1) / radius

    mob.rgbas = np.array(
        np.array(
            [interpolate(start_rgba, end_rgba, alpha) for alpha in alphas],
        ),
    )
    return self

def match_colors(self, mobject):
    Mobject.align_data(self, mobject)
    self.rgbas = np.array(mobject.rgbas)
    return self

def filter_out(self, condition):
    for mob in self.family_members_with_points():
        to_eliminate = ~np.apply_along_axis(condition, 1, mob.points)
        mob.points = mob.points[to_eliminate]
        mob.rgbas = mob.rgbas[to_eliminate]
    return self

def thin_out(self, factor=5):
    """
    Removes all but every nth point for n = factor
    """
    for mob in self.family_members_with_points():
        num_points = self.get_num_points()
        mob.apply_over_attr_arrays(
            lambda arr: arr[np.arange(0, num_points, factor)],
        )
    return self

def sort_points(self, function=lambda p: p[0]):
    """
    Function is any map from R^3 to R
    """
    for mob in self.family_members_with_points():
        indices = np.argsort(np.apply_along_axis(function, 1, mob.points))
        mob.apply_over_attr_arrays(lambda arr: arr[indices])
    return self

def fade_to(self, color, alpha, family=True):
    self.rgbas = interpolate(self.rgbas, color_to_rgba(color), alpha)
    for mob in self.submobjects:
        mob.fade_to(color, alpha, family)
    return self

def get_all_rgbas(self):
    return self.get_merged_array("rgbas")

def ingest_submobjects(self):
    attrs = self.get_array_attrs()
    arrays = list(map(self.get_merged_array, attrs))
    for attr, array in zip(attrs, arrays):
        setattr(self, attr, array)
    self.submobjects = []
    return self

def get_color(self):
    return rgba_to_color(self.rgbas[0, :])

def point_from_proportion(self, alpha):
    index = alpha * (self.get_num_points() - 1)
    return self.points[index]

@staticmethod
def get_mobject_type_class():
    return PMobject

# Alignment
def align_points_with_larger(self, larger_mobject):
    assert isinstance(larger_mobject, PMobject)
    self.apply_over_attr_arrays(
        lambda a: stretch_array_to_length(a, larger_mobject.get_num_points()),
    )

def get_point_mobject(self, center=None):
    if center is None:
        center = self.get_center()
    return Point(center)

def interpolate_color(self, mobject1, mobject2, alpha):
    self.rgbas = interpolate(mobject1.rgbas, mobject2.rgbas, alpha)
    self.set_stroke_width(
        interpolate(
            mobject1.get_stroke_width(),
            mobject2.get_stroke_width(),
            alpha,
        ),
    )
    return self

def pointwiseBecomePartial(self, mobject, a, b):
    lower_index, upper_index = (int(x * mobject.get_num_points()) for x in (a, b))
    for attr in self.get_array_attrs():
        full_array = getattr(mobject, attr)
        partial_array = full_array[lower_index:upper_index]
        setattr(self, attr, partial_array)

```

```

# TODO, Make the two implementations below non-redundant
class Mobject1D(PMobject, metaclass=ConvertToOpenGL):
    def __init__(self, density=DEFAULT_POINT_DENSITY_1D, **kwargs):
        self.density = density
        self.epsilon = 1.0 / self.density
        super().__init__(**kwargs)

    def add_line(self, start, end, color=None):
        start, end = list(map(np.array, [start, end]))
        length = np.linalg.norm(end - start)
        if length == 0:
            points = [start]
        else:
            epsilon = self.epsilon / length
            points = [interpolate(start, end, t) for t in np.arange(0, 1, epsilon)]
        self.add_points(points, color=color)

class Mobject2D(PMobject, metaclass=ConvertToOpenGL):
    def __init__(self, density=DEFAULT_POINT_DENSITY_2D, **kwargs):
        self.density = density
        self.epsilon = 1.0 / self.density
        super().__init__(**kwargs)

class PGroup(PMobject):
    """A group for several point mobjects.

    Examples
    -----
    .. manim:: PgroupExample
       :save_last_frame:

       class PgroupExample(Scene):
           def construct(self):
               p1 = PointCloudDot(radius=1, density=20, color=BLUE)
               p1.move_to(4.5 * LEFT)
               p2 = PointCloudDot()
               p3 = PointCloudDot(radius=1.5, stroke_width=2.5, color=PINK)
               p3.move_to(4.5 * RIGHT)
               pList = PGroup(p1, p2, p3)
               self.add(pList)

    """
    def __init__(self, *pmobs, **kwargs):
        if not all(isinstance(m, (PMobject, OpenGLMobject)) for m in pmobs):
            raise ValueError(
                "All submobjects must be of type PMobject or OpenGLMobject"
                " if using the opengl renderer",
            )
        super().__init__(**kwargs)
        self.add(*pmobs)

    def fade_to(self, color, alpha, family=True):
        if family:
            for mob in self.submobjects:
                mob.fade_to(color, alpha, family)

class PointCloudDot(Mobject1D):
    """A disc made of a cloud of dots.

    Examples
    -----
    .. manim:: PointCloudDotExample
       :save_last_frame:

       class PointCloudDotExample(Scene):
           def construct(self):
               cloud_1 = PointCloudDot(color=RED)
               cloud_2 = PointCloudDot(stroke_width=4, radius=1)
               cloud_3 = PointCloudDot(density=15)

               group = Group(cloud_1, cloud_2, cloud_3).arrange()
               self.add(group)

    """
    def __init__(self,
                 center=ORIGIN,
                 radius=2.0,
                 stroke_width=2,
                 density=DEFAULT_POINT_DENSITY_1D,
                 color=YELLOW,
                 **kwargs,
                 ):
        self.radius = radius
        self.epsilon = 1.0 / density
        super().__init__(
            stroke_width=stroke_width, density=density, color=color, **kwargs
        )
        self.shift(center)

    def init_points(self):
        self.reset_points()


```

```

    self.generate_points()

def generate_points(self):
    self.add_points(
        [
            r * (np.cos(theta) * RIGHT + np.sin(theta) * UP)
            for r in np.arange(self.epsilon, self.radius, self.epsilon)
            # Num is equal to int(stop - start)/ (step + 1) reformulated.
            for theta in np.linspace(
                0,
                2 * np.pi,
                num=int(2 * np.pi * (r + self.epsilon) / self.epsilon),
            )
        ],
    )

class Point(PMobject):
    """A mobject representing a point.

Examples
-----
.. manim:: ExamplePoint
    :save_last_frame:

    class ExamplePoint(Scene):
        def construct(self):
            colorList = [RED, GREEN, BLUE, YELLOW]
            for i in range(200):
                point = Point(location=[0.63 * np.random.randint(-4, 4), 0.37 * np.random.randint(-4, 4), 0], color=np.random.choice(colorList))
                self.add(point)
            for i in range(200):
                point = Point(location=[0.37 * np.random.randint(-4, 4), 0.63 * np.random.randint(-4, 4), 0], color=np.random.choice(colorList))
                self.add(point)
            self.add(point)

    def __init__(self, location=ORIGIN, color=BLACK, **kwargs):
        self.location = location
        super().__init__(color=color, **kwargs)

    def init_points(self):
        self.reset_points()
        self.generate_points()
        self.set_points([self.location])

    def generate_points(self):
        self.add_points([self.location])

```

opengl_renderer_window.py

```
from __future__ import annotations

import moderngl_window as mglw
from moderngl_window.context.pyglet.window import Window as PygletWindow
from moderngl_window.timers.clock import Timer
from screeninfo import get_monitors

from .. import __version__, config

class Window(PygletWindow):
    fullscreen = False
    resizable = True
    gl_version = (3, 3)
    vsync = True
    cursor = True

    def __init__(self, renderer, size=config.window_size, **kwargs):
        monitors = get_monitors()
        mon_index = config.window_monitor
        monitor = monitors[min(mon_index, len(monitors) - 1)]

        if size == "default":
            # make window_width half the width of the monitor
            # but make it full screen if --fullscreen

            window_width = monitor.width
            if not config.fullscreen:
                window_width //= 2

            # by default window_height = 9/16 * window_width
            window_height = int(
                window_width * config.frame_height // config.frame_width,
            )
            size = (window_width, window_height)
        else:
            size = tuple(size)

        super().__init__(size=size)

        self.title = f"Manim Community {__version__}"
        self.size = size
        self.renderer = renderer

        mglw.activate_context(window=self)
        self.timer = Timer()
        self.config = mglw.WindowConfig(ctx=self.ctx, wnd=self, timer=self.timer)
        self.timer.start()

        self.swap_buffers()

        initial_position = self.find_initial_position(size, monitor)
        self.position = initial_position

    # Delegate event handling to scene.
    def on_mouse_motion(self, x, y, dx, dy):
        super().on_mouse_motion(x, y, dx, dy)
        point = self.renderer.pixel_coords_to_space_coords(x, y)
        d_point = self.renderer.pixel_coords_to_space_coords(dx, dy, relative=True)
        self.renderer.scene.on_mouse_motion(point, d_point)

    def on_mouse_scroll(self, x, y, x_offset: float, y_offset: float):
        super().on_mouse_scroll(x, y, x_offset, y_offset)
        point = self.renderer.pixel_coords_to_space_coords(x, y)
        offset = self.renderer.pixel_coords_to_space_coords(
            x_offset,
            y_offset,
            relative=True,
        )
        self.renderer.scene.on_mouse_scroll(point, offset)

    def on_key_press(self, symbol, modifiers):
        self.renderer.pressed_keys.add(symbol)
        super().on_key_press(symbol, modifiers)
        self.renderer.scene.on_key_press(symbol, modifiers)

    def on_key_release(self, symbol, modifiers):
        if symbol in self.renderer.pressed_keys:
            self.renderer.pressed_keys.remove(symbol)
        super().on_key_release(symbol, modifiers)
        self.renderer.scene.on_key_release(symbol, modifiers)

    def on_mouse_drag(self, x, y, dx, dy, buttons, modifiers):
        super().on_mouse_drag(x, y, dx, dy, buttons, modifiers)
        point = self.renderer.pixel_coords_to_space_coords(x, y)
        d_point = self.renderer.pixel_coords_to_space_coords(dx, dy, relative=True)
        self.renderer.scene.on_mouse_drag(point, d_point, buttons, modifiers)
```

```

def find_initial_position(self, size, monitor):
    custom_position = config.window_position
    window_width, window_height = size
    # Position might be specified with a string of the form
    # x,y for integers x and y
    if len(custom_position) == 1:
        raise ValueError(
            "window_position must specify both Y and X positions (Y/X -> UR). Also accepts LEFT/RIGHT/ORIGIN/UP/DOWN.",
        )
    # in the form Y/X (UR)
    if custom_position in ["LEFT", "RIGHT"]:
        custom_position = "O" + custom_position[0]
    elif custom_position in ["UP", "DOWN"]:
        custom_position = custom_position[0] + "O"
    elif custom_position == "ORIGIN":
        custom_position = "O" * 2
    elif "," in custom_position:
        return tuple(map(int, custom_position.split(",")))

    # Alternatively, it might be specified with a string like
    # UR, OO, DL, etc. specifying what corner it should go to
    char_to_n = {"L": 0, "U": 0, "O": 1, "R": 2, "D": 2}
    width_diff = monitor.width - window_width
    height_diff = monitor.height - window_height

    return (
        monitor.x + char_to_n[custom_position[1]] * width_diff // 2,
        -monitor.y + char_to_n[custom_position[0]] * height_diff // 2,
    )

def on_mouse_press(self, x, y, button, modifiers):
    super().on_mouse_press(x, y, button, modifiers)
    point = self.renderer.pixel_coords_to_space_coords(x, y)
    mouse_button_map = {
        1: "LEFT",
        2: "MOUSE",
        4: "RIGHT",
    }
    self.renderer.scene.on_mouse_press(point, mouse_button_map[button], modifiers)

```

three_d_scene.py

```
"""A scene suitable for rendering three-dimensional objects and animations."""
from __future__ import annotations

__all__ = ["ThreeDScene", "SpecialThreeDScene"]

import warnings
from typing import Iterable, Sequence

import numpy as np

from manim.mobject.geometry.line import Line
from manim.mobject.graphing.coordinate_systems import ThreeDAxes
from manim.mobject.opengl.opengl_mobject import OpenGLMobject
from manim.mobject.three_d.three_dimensions import Sphere
from manim.mobject.value_tracker import ValueTracker

from .. import config
from ..animation.animation import Animation
from ..animation.transform import Transform
from ..camera.three_d_camera import ThreeDCamera
from ..constants import DEGREES, RendererType
from ..mobject.mobject import Mobject
from ..mobject.types.vectorized_mobject import VectorizedPoint, VGroup
from ..renderer.opengl_renderer import OpenGLCamera
from ..scene.scene import Scene
from ..utils.config_ops import merge_dicts_recursively

class ThreeDScene(Scene):
    """
    This is a Scene, with special configurations and properties that
    make it suitable for Three Dimensional Scenes.
    """

    def __init__(
        self,
        camera_class=ThreeDCamera,
        ambient_camera_rotation=None,
        default_angled_camera_orientation_kwargs=None,
        **kwargs,
    ):
        self.ambient_camera_rotation = ambient_camera_rotation
        if default_angled_camera_orientation_kwargs is None:
            default_angled_camera_orientation_kwargs = {
                "phi": 70 * DEGREES,
                "theta": -135 * DEGREES,
            }
        self.default_angled_camera_orientation_kwargs = (
            default_angled_camera_orientation_kwargs
        )
        super().__init__(camera_class=camera_class, **kwargs)

    def set_camera_orientation(
        self,
        phi: float | None = None,
        theta: float | None = None,
        gamma: float | None = None,
        zoom: float | None = None,
        focal_distance: float | None = None,
        frame_center: Mobject | Sequence[float] | None = None,
        **kwargs,
    ):
        """
        This method sets the orientation of the camera in the scene.

        Parameters
        -----
        phi
            The polar angle i.e the angle between Z_AXIS and Camera through ORIGIN in radians.

        theta
            The azimuthal angle i.e the angle that spins the camera around the Z_AXIS.

        focal_distance
            The focal_distance of the Camera.

        gamma
            The rotation of the camera about the vector from the ORIGIN to the Camera.
        
```

```

zoom
    The zoom factor of the scene.

frame_center
    The new center of the camera frame in cartesian coordinates.

"""

if phi is not None:
    self.renderer.camera.set_phi(phi)
if theta is not None:
    self.renderer.camera.set_theta(theta)
if focal_distance is not None:
    self.renderer.camera.set_focal_distance(focal_distance)
if gamma is not None:
    self.renderer.camera.set_gamma(gamma)
if zoom is not None:
    self.renderer.camera.set_zoom(zoom)
if frame_center is not None:
    self.renderer.camera._frame_center.move_to(frame_center)

def begin_ambient_camera_rotation(self, rate: float = 0.02, about: str = "theta"):
    """
    This method begins an ambient rotation of the camera about the Z_AXIS,
    in the anticlockwise direction

    Parameters
    -----
    rate
        The rate at which the camera should rotate about the Z_AXIS.
        Negative rate means clockwise rotation.
    about
        one of 3 options: ["theta", "phi", "gamma"]. defaults to theta.
    """
    # TODO, use a ValueTracker for rate, so that it
    # can begin and end smoothly
    about: str = about.lower()
    try:
        if config.renderer == RendererType.CAIRO:
            trackers = {
                "theta": self.camera.theta_tracker,
                "phi": self.camera.phi_tracker,
                "gamma": self.camera.gamma_tracker,
            }
            x: ValueTracker = trackers[about]
            x.add_updater(lambda m, dt: x.increment_value(rate * dt))
            self.add(x)
        elif config.renderer == RendererType.OPENGL:
            cam: OpenGLCamera = self.camera
            methods = {
                "theta": cam.increment_theta,
                "phi": cam.increment_phi,
                "gamma": cam.increment_gamma,
            }
            cam.add_updater(lambda m, dt: methods[about](rate * dt))
            self.add(self.camera)
    except Exception:
        raise ValueError("Invalid ambient rotation angle.")

def stop_ambient_camera_rotation(self, about="theta"):
    """
    This method stops all ambient camera rotation.
    """
    about: str = about.lower()
    try:
        if config.renderer == RendererType.CAIRO:
            trackers = {
                "theta": self.camera.theta_tracker,
                "phi": self.camera.phi_tracker,
                "gamma": self.camera.gamma_tracker,
            }
            x: ValueTracker = trackers[about]
            x.clear_updaters()
            self.remove(x)
        elif config.renderer == RendererType.OPENGL:
            self.camera.clear_updaters()
    except Exception:
        raise ValueError("Invalid ambient rotation angle.")

def begin_3dillusion_camera_rotation(
    self,
    rate: float = 1,
    origin_phi: float | None = None,
    origin_theta: float | None = None,
):
    """
    This method creates a 3D camera rotation illusion around

```

```

the current camera orientation.

Parameters
-----
rate
    The rate at which the camera rotation illusion should operate.
origin_phi
    The polar angle the camera should move around. Defaults
    to the current phi angle.
origin_theta
    The azimuthal angle the camera should move around. Defaults
    to the current theta angle.
"""
if origin_theta is None:
    origin_theta = self.renderer.camera.theta_tracker.get_value()
if origin_phi is None:
    origin_phi = self.renderer.camera.phi_tracker.get_value()

val_tracker_theta = ValueTracker(0)

def update_theta(m, dt):
    val_tracker_theta.increment_value(dt * rate)
    val_for_left_right = 0.2 * np.sin(val_tracker_theta.get_value())
    return m.set_value(origin_theta + val_for_left_right)

self.renderer.camera.theta_tracker.add_updater(update_theta)
self.add(self.renderer.camera.theta_tracker)

val_tracker_phi = ValueTracker(0)

def update_phi(m, dt):
    val_tracker_phi.increment_value(dt * rate)
    val_for_up_down = 0.1 * np.cos(val_tracker_phi.get_value()) - 0.1
    return m.set_value(origin_phi + val_for_up_down)

self.renderer.camera.phi_tracker.add_updater(update_phi)
self.add(self.renderer.camera.phi_tracker)

def stop_3dillusion_camera_rotation(self):
"""
This method stops all illusion camera rotations.
"""
self.renderer.camera.theta_tracker.clear_updaters()
self.remove(self.renderer.camera.theta_tracker)
self.renderer.camera.phi_tracker.clear_updaters()
self.remove(self.renderer.camera.phi_tracker)

def move_camera(
    self,
    phi: float | None = None,
    theta: float | None = None,
    gamma: float | None = None,
    zoom: float | None = None,
    focal_distance: float | None = None,
    frame_center: Mobject | Sequence[float] | None = None,
    added_anims: Iterable[Animation] = [],
    **kwargs,
):
"""
This method animates the movement of the camera
to the given spherical coordinates.

Parameters
-----
phi
    The polar angle i.e the angle between Z_AXIS and Camera through ORIGIN in radians.

theta
    The azimuthal angle i.e the angle that spins the camera around the Z_AXIS.

focal_distance
    The radial focal_distance between ORIGIN and Camera.

gamma
    The rotation of the camera about the vector from the ORIGIN to the Camera.

zoom
    The zoom factor of the camera.

frame_center
    The new center of the camera frame in cartesian coordinates.

added_anims
    Any other animations to be played at the same time.

"""
anims = []

```

```

if config.renderer == RendererType.CAIRO:
    self.camera: ThreeDCamera
    value_tracker_pairs = [
        (phi, self.camera.phi_tracker),
        (theta, self.camera.theta_tracker),
        (focal_distance, self.camera.focal_distance_tracker),
        (gamma, self.camera.gamma_tracker),
        (zoom, self.camera.zoom_tracker),
    ]
    for value, tracker in value_tracker_pairs:
        if value is not None:
            anims.append(tracker.animate.set_value(value))
    if frame_center is not None:
        anims.append(self.camera._frame_center.animate.move_to(frame_center))
elif config.renderer == RendererType.OPENGL:
    cam: OpenGLCamera = self.camera
    cam2 = cam.copy()
    methods = {
        "theta": cam2.set_theta,
        "phi": cam2.set_phi,
        "gamma": cam2.set_gamma,
        "zoom": cam2.scale,
        "frame_center": cam2.move_to,
    }
    if frame_center is not None:
        if isinstance(frame_center, OpenGLMobject):
            frame_center = frame_center.get_center()
        frame_center = list(frame_center)

    for value, method in [
        [theta, "theta"],
        [phi, "phi"],
        [gamma, "gamma"],
        [
            config.frame_height / (zoom * cam.height)
            if zoom is not None
            else None,
            "zoom",
        ],
        [frame_center, "frame_center"],
    ]:
        if value is not None:
            methods[method](value)

    if focal_distance is not None:
        warnings.warn(
            "focal distance of OpenGLCamera can not be adjusted.",
            stacklevel=2,
        )

    anims += [Transform(cam, cam2)]

self.play(*anims + added_anims, **kwargs)

# These lines are added to improve performance. If manim thinks that frame_center is moving,
# it is required to redraw every object. These lines remove frame_center from the Scene once
# its animation is done, ensuring that manim does not think that it is moving. Since the
# frame_center is never actually drawn, this shouldn't break anything.
if frame_center is not None and config.renderer == RendererType.CAIRO:
    self.remove(self.camera._frame_center)

def get_moving_mobjects(self, *animations: Animation):
    """
    This method returns a list of all of the Mobjects in the Scene that
    are moving, that are also in the animations passed.

    Parameters
    -----
    *animations
        The animations whose mobjects will be checked.
    """
    moving_mobjects = super().get_moving_mobjects(*animations)
    camera_mobjects = self.renderer.camera.get_value_trackers() + [
        self.renderer.camera._frame_center,
    ]
    if any(cm in moving_mobjects for cm in camera_mobjects):
        return self.mobjects
    return moving_mobjects

def add_fixed_orientation_mobjects(self, *mobjects: Mobject, **kwargs):
    """
    This method is used to prevent the rotation and tilting
    of mobjects as the camera moves around. The mobject can
    still move in the x,y,z directions, but will always be
    at the angle (relative to the camera) that it was at
    when it was passed through this method.)
    """

```

```

Parameters
-----
*mobjects
    The Mobject(s) whose orientation must be fixed.

**kwargs
    Some valid kwargs are
        use_static_center_func : bool
        center_func : function
"""

if config.renderer == RendererType.CAIRO:
    self.add(*mobjects)
    self.renderer.camera.add_fixed_orientation_mobjects(*mobjects, **kwargs)
elif config.renderer == RendererType.OPENGL:
    for mob in mobjects:
        mob: OpenGLMobject
        mob.fix_orientation()
        self.add(mob)

def add_fixed_in_frame_mobjects(self, *mobjects: Mobject):
"""
This method is used to prevent the rotation and movement
of mobjects as the camera moves around. The mobject is
essentially overlaid, and is not impacted by the camera's
movement in any way.

Parameters
-----
*mobjects
    The Mobjects whose orientation must be fixed.
"""

if config.renderer == RendererType.CAIRO:
    self.add(*mobjects)
    self.camera: ThreeDCamera
    self.camera.add_fixed_in_frame_mobjects(*mobjects)
elif config.renderer == RendererType.OPENGL:
    for mob in mobjects:
        mob: OpenGLMobject
        mob.fix_in_frame()
        self.add(mob)

def remove_fixed_orientation_mobjects(self, *mobjects: Mobject):
"""
This method "unfixes" the orientation of the mobjects
passed, meaning they will no longer be at the same angle
relative to the camera. This only makes sense if the
mobject was passed through add_fixed_orientation_mobjects first.

Parameters
-----
*mobjects
    The Mobjects whose orientation must be unfixed.
"""

if config.renderer == RendererType.CAIRO:
    self.renderer.camera.remove_fixed_orientation_mobjects(*mobjects)
elif config.renderer == RendererType.OPENGL:
    for mob in mobjects:
        mob: OpenGLMobject
        mob.unfix_orientation()
        self.remove(mob)

def remove_fixed_in_frame_mobjects(self, *mobjects: Mobject):
"""
This method undoes what add_fixed_in_frame_mobjects does.
It allows the mobject to be affected by the movement of
the camera.

Parameters
-----
*mobjects
    The Mobjects whose position and orientation must be unfixed.
"""

if config.renderer == RendererType.CAIRO:
    self.renderer.camera.remove_fixed_in_frame_mobjects(*mobjects)
elif config.renderer == RendererType.OPENGL:
    for mob in mobjects:
        mob: OpenGLMobject
        mob.unfix_from_frame()
        self.remove(mob)

## def set_to_default_angled_camera_orientation(self, **kwargs):
"""
This method sets the default_angled_camera_orientation to the
keyword arguments passed, and sets the camera to that orientation.

```

```

Parameters
-----
**kwargs
    Some recognised kwargs are phi, theta, focal_distance, gamma,
    which have the same meaning as the parameters in set_camera_orientation.
"""
config = dict(
    self.default_camera_orientation_kwargs,
) # Where doe this come from?
config.update(kwargs)
self.set_camera_orientation(**config)

class SpecialThreeDScene(ThreeDScene):
    """An extension of :class:`ThreeDScene` with more settings.

    It has some extra configuration for axes, spheres,
    and an override for low quality rendering. Further key differences
    are:

    * The camera shades applicable 3DObjects by default,
      except if rendering in low quality.
    * Some default params for Spheres and Axes have been added.

    """
    def __init__(
        self,
        cut_axes_at_radius=True,
        camera_config={"should_apply_shading": True, "exponential_projection": True},
        three_d_axes_config={
            "num_axis_pieces": 1,
            "axis_config": {
                "unit_size": 2,
                "tick_frequency": 1,
                "numbers_with_elongated_ticks": [0, 1, 2],
                "stroke_width": 2,
            },
        },
        sphere_config={"radius": 2, "resolution": (24, 48)},
        default_angled_camera_position={
            "phi": 70 * DEGREES,
            "theta": -110 * DEGREES,
        },
        # When scene is extracted with -l flag, this
        # configuration will override the above configuration.
        low_quality_config={
            "camera_config": {"should_apply_shading": False},
            "three_d_axes_config": {"num_axis_pieces": 1},
            "sphere_config": {"resolution": (12, 24)},
        },
        **kwargs,
    ):
        self.cut_axes_at_radius = cut_axes_at_radius
        self.camera_config = camera_config
        self.three_d_axes_config = three_d_axes_config
        self.sphere_config = sphere_config
        self.default_angled_camera_position = default_angled_camera_position
        self.low_quality_config = low_quality_config
        if self.renderer.camera_config["pixel_width"] == config["pixel_width"]:
            _config = {}
        else:
            _config = self.low_quality_config
        _config = merge_dicts_recursively(_config, kwargs)
        super().__init__(**_config)

    def get_axes(self):
        """Return a set of 3D axes.

        Returns
        -----
        :class:`.ThreeDAxes`
            A set of 3D axes.
        """
        axes = ThreeDAxes(**self.three_d_axes_config)
        for axis in axes:
            if self.cut_axes_at_radius:
                p0 = axis.get_start()
                p1 = axis.number_to_point(-1)
                p2 = axis.number_to_point(1)
                p3 = axis.get_end()
                new_pieces = VGroup(Line(p0, p1), Line(p1, p2), Line(p2, p3))
                for piece in new_pieces:
                    piece.shade_in_3d = True
                new_pieces.match_style(axis.pieces)
                axis.pieces.subobjects = new_pieces.subobjects
            for tick in axis.tick_marks:

```

```
        tick.add(VectorizedPoint(1.5 * tick.get_center())))
    return axes

def get_sphere(self, **kwargs):
    """
    Returns a sphere with the passed keyword arguments as properties.

    Parameters
    -----
    **kwargs
        Any valid parameter of :class:`~.Sphere` or :class:`~.Surface`.

    Returns
    -----
    :class:`~.Sphere`
        The sphere object.
    """
    config = merge_dicts_recursively(self.sphere_config, kwargs)
    return Sphere(**config)

def get_default_camera_position(self):
    """
    Returns the default_angled_camera position.

    Returns
    -----
    dict
        Dictionary of phi, theta, focal_distance, and gamma.
    """
    return self.default_angled_camera_position

def set_camera_to_default_position(self):
    """
    Sets the camera to its default position.

    """
    self.set_camera_orientation(**self.default_angled_camera_position)
```

opengl_compatibility.py

```
from __future__ import annotations

from abc import ABCMeta

from manim import config
from manim.mobject.opengl.opengl_mobject import OpenGLMobject
from manim.mobject.opengl.opengl_point_cloud_mobject import OpenGLPMobject
from manim.mobject.opengl.opengl_three_dimensions import OpenGLSurface
from manim.mobject.opengl.opengl_vectorized_mobject import OpenGLVMOobject

from ...constants import RendererType


class ConvertToOpenGL(ABCMeta):
    """Metaclass for swapping (V)Mobject with its OpenGL counterpart at runtime
    depending on config.renderer. This metaclass should only need to be inherited
    on the lowest order inheritance classes such as Mobject and VMOobject.
    """
    _converted_classes = []

    def __new__(mcls, name, bases, namespace): # noqa: B902
        if config.renderer == RendererType.OPENGL:
            # Must check class names to prevent
            # cyclic importing.
            base_names_to_opengl = {
                "Mobject": OpenGLMobject,
                "VMOobject": OpenGLVMOobject,
                "PMobject": OpenGLPMobject,
                "Mobject1D": OpenGLMobject,
                "Mobject2D": OpenGLMobject,
                "Surface": OpenGLSurface,
            }
            bases = tuple(
                base_names_to_opengl.get(base.__name__, base) for base in bases
            )
        return super().__new__(mcls, name, bases, namespace)

    def __init__(cls, name, bases, namespace): # noqa: B902
        super().__init__(name, bases, namespace)
        cls._converted_classes.append(cls)
```

default_group.py

```
"""DefaultGroup allows a subcommand to act as the main command

In particular, this class is what allows ``manim`` to act as ``manim render``.

"""

import cloup

from .. import logger

__all__ = ["DefaultGroup"]


class DefaultGroup(cloup.Group):
    """Invokes a subcommand marked with ``default=True`` if any subcommand not chosen.

    """

    def __init__(self, *args, **kwargs):
        # To resolve as the default command.
        if not kwargs.get("ignore_unknown_options", True):
            raise ValueError("Default group accepts unknown options")
        self.ignore_unknown_options = True
        self.default_cmd_name = kwargs.pop("default", None)
        self.default_if_no_args = kwargs.pop("default_if_no_args", False)
        super().__init__(*args, **kwargs)

    def set_default_command(self, command):
        """Sets a command function as the default command."""
        cmd_name = command.name
        self.add_command(command)
        self.default_cmd_name = cmd_name

    def parse_args(self, ctx, args):
        if not args and self.default_if_no_args:
            args.insert(0, self.default_cmd_name)
        return super().parse_args(ctx, args)

    def get_command(self, ctx, cmd_name):
        if cmd_name not in self.commands:
            # No command name matched.
            ctx.arg0 = cmd_name
            cmd_name = self.default_cmd_name
        return super().get_command(ctx, cmd_name)

    def resolve_command(self, ctx, args):
        base = super()
        cmd_name, cmd, args = base.resolve_command(ctx, args)
        if hasattr(ctx, "arg0"):
            args.insert(0, ctx.arg0)
            cmd_name = cmd.name
        return cmd_name, cmd, args

    def command(self, *args, **kwargs):
        default = kwargs.pop("default", False)
        decorator = super().command(*args, **kwargs)
        if not default:
            return decorator
        logger.log(
            "Use default param of DefaultGroup or \"set_default_command() instead",
            DeprecationWarning,
        )

        def _decorator(f):
            cmd = decorator(f)
            self.set_default_command(cmd)
            return cmd

        return _decorator
```

module_ops.py

```

from __future__ import annotations

import importlib.util
import inspect
import os
import re
import sys
import types
import warnings
from pathlib import Path

from .. import config, console, constants, logger
from ..scene.scene_file_writer import SceneFileWriter

def get_module(file_name: Path):
    if str(file_name) == "-":
        module = types.ModuleType("input_scenes")
        logger.info(
            "Enter the animation's code & end with an EOF (CTRL+D on Linux/Unix, CTRL+Z on Windows):"
        )
        code = sys.stdin.read()
        if not code.startswith("from manim import"):
            logger.warning(
                "Didn't find an import statement for Manim. Importing automatically...",
            )
        code = "from manim import *\n" + code
        logger.info("Rendering animation from typed code...")
    try:
        exec(code, module.__dict__)
        return module
    except Exception as e:
        logger.error(f"Failed to render scene: {str(e)}")
        sys.exit(2)
    else:
        if file_name.exists():
            ext = file_name.suffix
            if ext != ".py":
                raise ValueError(f"{file_name} is not a valid Manim python script.")
            module_name = "..".join(file_name.with_suffix("").parts)

            warnings.filterwarnings(
                "default",
                category=DeprecationWarning,
                module=module_name,
            )

            spec = importlib.util.spec_from_file_location(module_name, file_name)
            module = importlib.util.module_from_spec(spec)
            sys.modules[module_name] = module
            sys.path.insert(0, str(file_name.parent.absolute()))
            spec.loader.exec_module(module)
            return module
        else:
            raise FileNotFoundError(f"{file_name} not found")

def get_scene_classes_from_module(module):
    from ..scene.scene import Scene

    def is_child_scene(obj, module):
        return (
            inspect.isclass(obj)
            and issubclass(obj, Scene)
            and obj != Scene
            and obj.__module__.startswith(module.__name__)
        )

    return [
        member[1]
        for member in inspect.getmembers(module, lambda x: is_child_scene(x, module))
    ]

def get_scenes_to_render(scene_classes):
    if not scene_classes:
        logger.error(constants.NO_SCENE_MESSAGE)
        return []
    if config["write_all"]:
        return scene_classes
    result = []

```

```

for scene_name in config["scene_names"]:
    found = False
    for scene_class in scene_classes:
        if scene_class.__name__ == scene_name:
            result.append(scene_class)
            found = True
            break
    if not found and (scene_name != ""):
        logger.error(constants.SCENE_NOT_FOUND_MESSAGE.format(scene_name))
if result:
    return result
if len(scene_classes) == 1:
    config["scene_names"] = [scene_classes[0].__name__]
    return [scene_classes[0]]
return prompt_user_for_choice(scene_classes)

def prompt_user_for_choice(scene_classes):
    num_to_class = {}
    SceneFileWriter.force_output_as_scene_name = True
    for count, scene_class in enumerate(scene_classes, 1):
        name = scene_class.__name__
        console.print(f"{count}: {name}", style="logging.level.info")
        num_to_class[count] = scene_class
    try:
        user_input = console.input(
            f"[log.message] {constants.CHOOSE_NUMBER_MESSAGE} [/log.message]",
        )
        scene_classes = [
            num_to_class[int(num_str)]
            for num_str in re.split(r"\s*,\s*", user_input.strip())
        ]
        config["scene_names"] = [scene_class.__name__ for scene_class in scene_classes]
        return scene_classes
    except KeyError:
        logger.error(constants.INVALID_NUMBER_MESSAGE)
        sys.exit(2)
    except EOFError:
        sys.exit(1)
    except ValueError:
        logger.error("No scenes were selected. Exiting.")
        sys.exit(1)

def scene_classes_from_file(
    file_path: Path, require_single_scene=False, full_list=False
):
    module = get_module(file_path)
    all_scene_classes = get_scene_classes_from_module(module)
    if full_list:
        return all_scene_classes
    scene_classes_to_render = get_scenes_to_render(all_scene_classes)
    if require_single_scene:
        assert len(scene_classes_to_render) == 1
        return scene_classes_to_render[0]
    return scene_classes_to_render

```

cli_colors.py

```
import configparser

from cloup import Context, HelpFormatter, HelpTheme, Style

def parse_cli_ctx(parser: configparser.ConfigParser) -> Context:
    formatter_settings = {
        "indent_increment": int(parser["indent_increment"]),
        "width": int(parser["width"]),
        "coll_max_width": int(parser["coll_max_width"]),
        "col2_min_width": int(parser["col2_min_width"]),
        "col_spacing": int(parser["col_spacing"]),
        "row_sep": parser["row_sep"] if parser["row_sep"] else None,
    }
    theme_settings = {}
    theme_keys = [
        "command_help",
        "invoked_command",
        "heading",
        "constraint",
        "section_help",
        "coll",
        "col2",
        "epilog",
    ]
    for k, v in parser.items():
        if k in theme_keys and v:
            theme_settings.update({k: Style(v)})
    formatter = {}
    theme = parser["theme"] if parser["theme"] else None
    if theme is None:
        formatter = HelpFormatter().settings(
            theme=HelpTheme(**theme_settings), **formatter_settings
        )
    elif theme.lower() == "dark":
        formatter = HelpFormatter().settings(
            theme=HelpTheme.dark().with_(**theme_settings), **formatter_settings
        )
    elif theme.lower() == "light":
        formatter = HelpFormatter().settings(
            theme=HelpTheme.light().with_(**theme_settings), **formatter_settings
        )
    return Context.settings(
        align_option_groups=parser["align_option_groups"].lower() == "true",
        align_sections=parser["align_sections"].lower() == "true",
        show_constraints=True,
        formatter_settings=formatter,
    )
```

scene.py

```
"""Basic canvas for animations."""

from __future__ import annotations

__all__ = ["Scene"]

import copy
import datetime
import inspect
import platform
import random
import threading
import time
import types
from queue import Queue
from typing import Callable

import srt

from manim.scene.section import DefaultSectionType

try:
    import dearpygui.dearpygui as dpg

    dearpygui_imported = True
except ImportError:
    dearpygui_imported = False
import numpy as np
from tqdm import tqdm
from watchdog.events import FileSystemEventHandler
from watchdog.observers import Observer

from manim.mobject.mobject import Mobject
from manim.mobject.opengl_mobject import OpenGLPoint

from .. import config, logger
from ..animation.animation import Animation, Wait, prepare_animation
from ..camera.camera import Camera
from ..constants import *
from ..gui.gui import configure_pygui
from ..renderer.cairo_renderer import CairoRenderer
from ..renderer.opengl_renderer import OpenGLRenderer
from ..renderer.shader import Object3D
from ..utils import opengl, space_ops
from ..utils.exceptions import EndSceneEarlyException, RerunSceneException
from ..utils.family import extract_mobject_family_members
from ..utils.family_ops import restructure_list_to_exclude_certain_family_members
from ..utils.file_ops import open_media_file
from ..utils.iterables import list_difference_update, list_update

class RerunSceneHandler(FileSystemEventHandler):
    """A class to handle rerunning a Scene after the input file is modified."""

    def __init__(self, queue):
        super().__init__()
        self.queue = queue

    def on_modified(self, event):
        self.queue.put(("rerun_file", [], {}))

class Scene:
    """A Scene is the canvas of your animation.

    The primary role of :class:`Scene` is to provide the user with tools to manage
    mobjects and animations. Generally speaking, a manim script consists of a class
    that derives from :class:`Scene` whose :meth:`Scene.construct` method is overridden
    by the user's code.

    Mobjects are displayed on screen by calling :meth:`Scene.add` and removed from
    screen by calling :meth:`Scene.remove`. All mobjects currently on screen are kept
    in :attr:`Scene.mobjects`. Animations are played by calling :meth:`Scene.play`.

    A :class:`Scene` is rendered internally by calling :meth:`Scene.render`. This in
    turn calls :meth:`Scene.setup`, :meth:`Scene.construct`, and
    :meth:`Scene.tear_down`, in that order.

    It is not recommended to override the ``__init__`` method in user Scenes. For code
    that should be ran before a Scene is rendered, use :meth:`Scene.setup` instead.

    Examples
    -----
    Override the :meth:`Scene.construct` method with your code.

    .. code-block:: python

        class MyScene(Scene):
            def construct(self):
                self.play(Write(Text("Hello World!")))

    """

    def __init__(
        self,
        renderer=None,
        camera_class=Camera,
        always_update_mobjects=False,
        random_seed=None,
        skip_animations=False,
    ):
        self.camera_class = camera_class
        self.always_update_mobjects = always_update_mobjects
        self.random_seed = random_seed
        self.skip_animations = skip_animations

        self.animations = None
        self.stop_condition = None
        self.moving_mobjects = []
        self.static_mobjects = []
        self.time_progression = None
        self.duration = None
        self.last_t = None
        self.queue = Queue()
        self.skip_animation_preview = False
```

```

self.meshes = []
self.camera_target = ORIGIN
self.widgets = []
self.dearpygui_imported = dearpygui_imported
self.updaters = []
self.point_lights = []
self.ambient_light = None
self.key_to_function_map = {}
self.mouse_press_callbacks = []
self.interactive_mode = False

if config.renderer == RendererType.OPENGL:
    # Items associated with interaction
    self.mouse_point = OpenGLPoint()
    self.mouse_drag_point = OpenGLPoint()
    if renderer is None:
        renderer = OpenGLRenderer()

if renderer is None:
    self.renderer = CairoRenderer(
        camera_class=self.camera_class,
        skip_animations=self.skip_animations,
    )
else:
    self.renderer = renderer
self.renderer.init_scene(self)

self.mobjects = []
# TODO, remove need for foreground mobjects
self.foreground_mobjects = []
if self.random_seed is not None:
    random.seed(self.random_seed)
    np.random.seed(self.random_seed)

@property
def camera(self):
    return self.renderer.camera

def __deepcopy__(self, clone_from_id):
    cls = self.__class__
    result = cls.__new__(cls)
    clone_from_id[id(self)] = result
    for k, v in self.__dict__.items():
        if k in ["renderer", "time_progression"]:
            continue
        if k == "camera_class":
            setattr(result, k, v)
        setattr(result, k, copy.deepcopy(v, clone_from_id))
    result.mobject_updater_lists = []

    # Update updaters
    for mobject in self.mobjects:
        cloned_updaters = []
        for updater in mobject.updaters:
            # Make the cloned updater use the cloned Mobjects as free variables
            # rather than the original ones. Analyzing function bytecode with the
            # dis module will help in understanding this.
            # https://docs.python.org/3/library/dis.html
            # TODO: Do the same for function calls recursively.
            free_variable_map = inspect.getclosurevars(updater).nonlocals
            cloned_co_freevars = []
            cloned_closure = []
            for free_variable_name in updater.__code__.co_freevars:
                free_variable_value = free_variable_map[free_variable_name]

                # If the referenced variable has not been cloned, raise.
                if id(free_variable_value) not in clone_from_id:
                    raise Exception(
                        f'{free_variable_name} is referenced from an updater '
                        'but is not an attribute of the Scene, which isn\'t '
                        'allowed.',
                    )

                # Add the cloned object's name to the free variable list.
                cloned_co_freevars.append(free_variable_name)

            # Add a cell containing the cloned object's reference to the
            # closure list.
            cloned_closure.append(
                types.CellType(clone_from_id[id(free_variable_value)]),
            )

            cloned_updater = types.FunctionType(
                updater.__code__.replace(co_freevars=tuple(cloned_co_freevars)),
                updater.__globals__,
                updater.__name__,
                updater.__defaults__,
                tuple(cloned_closure),
            )
            cloned_updaters.append(cloned_updater)
        mobject_clone = clone_from_id[id(mobject)]
        mobject_clone.updaters = cloned_updaters
        if len(cloned_updaters) > 0:
            result.mobject_updater_lists.append((mobject_clone, cloned_updaters))
    return result

def render(self, preview: bool = False):
    """
    Renders this Scene.

    Parameters
    -----
    preview
        If true, opens scene in a file viewer.
    """
    self.setup()
    try:
        self.construct()
    except EndSceneEarlyException:
        pass
    except RerunSceneException as e:
        self.remove("self.mobjects")
        self.renderer.clear_screen()
        self.renderer.num_plays = 0
        return True
    self.tear_down()
    # We have to reset these settings in case of multiple renders.
    self.renderer.scene_finished(self)

    # Show info only if animations are rendered or to get image
    if (

```

```

        self.renderer.num_plays
        or config["format"] == "png"
        or config["save_last_frame"]
    ):
        logger.info(
            f"Rendered {str(self)}\nPlayed {self.renderer.num_plays} animations",
        )

    # If preview open up the render after rendering.
    if preview:
        config["preview"] = True

    if config["preview"] or config["show_in_file_browser"]:
        open_media_file(self.renderer.file_writer)

def setup(self):
    """
    This is meant to be implemented by any scenes which
    are commonly subclassed, and have some common setup
    involved before the construct method is called.
    """
    pass

def tear_down(self):
    """
    This is meant to be implemented by any scenes which
    are commonly subclassed, and have some common method
    to be invoked before the scene ends.
    """
    pass

def construct(self):
    """
    Add content to the Scene.

    From within :meth:`Scene.construct`, display mobjects on screen by calling
    :meth:`Scene.add` and remove them from screen by calling :meth:`Scene.remove`.
    All mobjects currently on screen are kept in :attr:`Scene.mobjects`. Play
    animations by calling :meth:`Scene.play`.

    Notes
    -----
    Initialization code should go in :meth:`Scene.setup`. Termination code should
    go in :meth:`Scene.tear_down`.

    Examples
    -----
    A typical manim script includes a class derived from :class:`Scene` with an
    overridden :meth:`Scene.construct` method:

    .. code-block:: python

        class MyScene(Scene):
            def construct(self):
                self.play(Write(Text("Hello World!")))

    See Also
    -----
    :meth:`Scene.setup`
    :meth:`Scene.render`
    :meth:`Scene.tear_down`

    """
    pass # To be implemented in subclasses

def next_section(
    self,
    name: str = "unnamed",
    type: str = DefaultSectionType.NORMAL,
    skip_animations: bool = False,
) -> None:
    """
    Create separation here; the last section gets finished and a new one gets created.
    ``skip_animations`` skips the rendering of all animations in this section.
    Refer to :doc:`the documentation</tutorials/output_and_config>` on how to use sections.
    """
    self.renderer.file_writer.next_section(name, type, skip_animations)

def __str__(self):
    return self.__class__.__name__

def get_attrs(self, *keys: str):
    """
    Gets attributes of a scene given the attribute's identifier/name.

    Parameters
    -----
    *keys
        Name(s) of the argument(s) to return the attribute of.

    Returns
    -----
    list
        List of attributes of the passed identifiers.
    """
    return [getattr(self, key) for key in keys]

def update_mobjects(self, dt: float):
    """
    Begins updating all mobjects in the Scene.

    Parameters
    -----
    dt
        Change in time between updates. Defaults (mostly) to 1/frames_per_second
    """
    for mobject in self.mobjects:
        mobject.update(dt)

def update_meshes(self, dt):
    for obj in self.meshes:
        for mesh in obj.get_family():
            mesh.update(dt)

def update_self(self, dt: float):
    """
    Run all scene updater functions.

    Among all types of update functions (mobject updaters, mesh updaters,
    scene updaters), scene update functions are called last.

    Parameters
    -----
    dt
    
```

```

Scene time since last update.

See Also
-----
:meth:`Scene.add_updater`
:meth:`Scene.remove_updater`
"""
for func in self.updaters:
    func(dt)

def should_update_mobjects(self) -> bool:
    """
    Returns True if the mobjects of this scene should be updated.

    In particular, this checks whether
    - the :attr:`always_update_mobjects` attribute of :class:`.Scene` is set to ``True``,
    - the :class:`.Scene` itself has time-based updaters attached,
    - any mobject in this :class:`.Scene` has time-based updaters attached.

    This is only called when a single Wait animation is played.
    """
    wait_animation = self.animations[0]
    if wait_animation.is_static_wait is None:
        should_update = (
            self.always_update_mobjects
            or self.updaters
            or wait_animation.stop_condition is not None
            or any(
                mob.has_time_based_updater()
                for mob in self.get_mobject_family_members()
            )
        )
        wait_animation.is_static_wait = not should_update
    return not wait_animation.is_static_wait

def get_top_level_mobjects(self):
    """
    Returns all mobjects which are not submobjects.

    Returns
    -----
    list
        List of top level mobjects.
    """
    # Return only those which are not in the family
    # of another mobject from the scene
    families = [m.get_family() for m in self.mobjects]

    def is_top_level(mobject):
        num_families = sum((mobject in family) for family in families)
        return num_families == 1

    return list(filter(is_top_level, self.mobjects))

def get_mobject_family_members(self):
    """
    Returns list of family-members of all mobjects in scene.
    If a Circle() and a VGroup(Rectangle(), Triangle()) were added, it returns not only the Circle(), Rectangle() and Triangle(), but also the VGroup() object.

    Returns
    -----
    list
        List of mobject family members.
    """
    if config.renderer == RendererType.OPENGL:
        family_members = []
        for mob in self.mobjects:
            family_members.extend(mob.get_family())
        return family_members
    elif config.renderer == RendererType.CAIRO:
        return extract_mobject_family_members(
            self.mobjects,
            use_z_index=config.renderer.camera.use_z_index,
        )

def add(self, *mobjects: Mobject):
    """
    Mobjects will be displayed, from background to foreground in the order with which they are added.

    Parameters
    -----
    *mobjects
        Mobjects to add.

    Returns
    -----
    Scene
        The same scene after adding the Mobjects in.

    """
    if config.renderer == RendererType.OPENGL:
        new_mobjects = []
        new_meshes = []
        for mobject_or_mesh in mobjects:
            if isinstance(mobject_or_mesh, Object3D):
                new_meshes.append(mobject_or_mesh)
            else:
                new_mobjects.append(mobject_or_mesh)
        self.remove(*new_mobjects)
        self.mobjects += new_mobjects
        self.remove(*new_meshes)
        self.meshes += new_meshes
    elif config.renderer == RendererType.CAIRO:
        mobjects = [*mobjects, *self.foreground_mobjects]
        self.restructure_mobjects(to_remove=mobjects)
        self.mobjects += mobjects
        if self.moving_mobjects:
            self.restructure_mobjects(
                to_remove=mobjects,
                mobject_list_name="moving_mobjects",
            )
            self.moving_mobjects += mobjects
    return self

def add_mobjects_from_animations(self, animations):
    curr_mobjects = self.get_mobject_family_members()

```

```

for animation in animations:
    if animation.is_introducer():
        continue
    # Anything animated that's not already in the
    # scene gets added to the scene
    mob = animation.mobject
    if mob is not None and mob not in curr_mobjects:
        self.add(mob)
        curr_mobjects += mob.get_family()

def remove(self, *mobjects: Mobject):
    """
    Removes mobjects in the passed list of mobjects
    from the scene and the foreground, by removing them
    from "mobjects" and "foreground_mobjects"

    Parameters
    -----
    *mobjects
        The mobjects to remove.
    """
    if config.renderer == RendererType.OPENGL:
        mobjects_to_remove = []
        meshes_to_remove = set()
        for mobject_or_mesh in mobjects:
            if isinstance(mobject_or_mesh, Object3D):
                meshes_to_remove.add(mobject_or_mesh)
            else:
                mobjects_to_remove.append(mobject_or_mesh)
        self.mobjects = restructure_list_to_exclude_certain_family_members(
            self.mobjects,
            mobjects_to_remove,
        )
        self.meshes = list(
            filter(lambda mesh: mesh not in set(meshes_to_remove), self.meshes),
        )
    return self
    elif config.renderer == RendererType.CAIRO:
        for list_name in "mobjects", "foreground_mobjects":
            self.restructure_mobjects(mobjects, list_name, False)
        return self

def replace(self, old_mobject: Mobject, new_mobject: Mobject) -> None:
    """
    Replace one mobject in the scene with another, preserving draw order.

    If ``old_mobject`` is a submobject of some other Mobject (e.g. a
    :class:`.Group`), the new mobject will replace it inside the group,
    without otherwise changing the parent mobject.

    Parameters
    -----
    old_mobject
        The mobject to be replaced. Must be present in the scene.
    new_mobject
        A mobject which must not already be in the scene.

    """
    if old_mobject is None or new_mobject is None:
        raise ValueError("Specified mobjects cannot be None")

    def replace_in_list(
        mobj_list: list[Mobject], old_m: Mobject, new_m: Mobject
    ) -> bool:
        # We use breadth-first search because some Mobjects get very deep and
        # we expect top-level elements to be the most common targets for replace.
        for i in range(0, len(mobj_list)):
            # Is this the old mobject?
            if mobj_list[i] == old_m:
                # If so, write the new object to the same spot and stop looking.
                mobj_list[i] = new_m
                return True
        # Now check all the children of all these mobs.
        for mob in mobj_list: # noqa: SIM110
            if replace_in_list(mob.submobjects, old_m, new_m):
                # If we found it in a submobject, stop looking.
                return True
        # If we did not find the mobject in the mobject list or any submobjects,
        # (or the list was empty), indicate we did not make the replacement.
        return False

    # Make use of short-circuiting conditionals to check mobjects and then
    # foreground_mobjects
    replaced = replace_in_list(
        self.mobjects, old_mobject, new_mobject
    ) or replace_in_list(self.foreground_mobjects, old_mobject, new_mobject)

    if not replaced:
        raise ValueError(f"Could not find {old_mobject} in scene")

def add_updater(self, func: Callable[[float], None]) -> None:
    """
    Add an update function to the scene.

    The scene updater functions are run every frame,
    and they are the last type of updaters to run.

    ... WARNING::

        When using the Cairo renderer, scene updaters that
        modify mobjects are not detected in the same way
        that mobject updaters are. To be more concrete,
        a mobject only modified via a scene updater will
        not necessarily be added to the list of *moving*
        mobjects* and thus might not be updated every frame.

        TL;DR: Use mobject updaters to update mobjects.

    Parameters
    -----
    func
        The updater function. It takes a float, which is the
        time difference since the last update (usually equal
        to the frame rate).

    See also
    -----
    :meth:`Scene.remove_updater`
    :meth:`Scene.update_self`
    """
    self.updaters.append(func)

def remove_updater(self, func: Callable[[float], None]) -> None:

```

```

"""Remove an update function from the scene.

Parameters
-----
func
    The updater function to be removed.

See also
-----
:meth:`Scene.add_updater`
:meth:`Scene.update_self`
"""

self.updaters = [f for f in self.updaters if f is not func]

def restructure_mobjects(
    self,
    to_remove: Mobject,
    mobject_list_name: str = "mobjects",
    extract_families: bool = True,
):
    """
    If your scene has a Group(), and you removed a mobject from the Group,
    this dissolves the group and puts the rest of the mobjects directly
    in self.mobjects or self.foreground_mobjects.

    In cases where the scene contains a group, e.g. Group(m1, m2, m3), but one
    of its submobjects is removed, e.g. scene.remove(m1), the list of mobjects
    will be edited to contain other submobjects, but not m1, e.g. it will now
    insert m2 and m3 to where the group once was.

    Parameters
    -----
    to_remove
        The Mobject to remove.

    mobject_list_name
        The list of mobjects ("mobjects", "foreground_mobjects" etc) to remove from.

    extract_families
        Whether the mobject's families should be recursively extracted.

    Returns
    -----
    Scene
        The Scene mobject with restructured Mobjects.
    """

    if extract_families:
        to_remove = extract_mobject_family_members(
            to_remove,
            use_z_index=self.renderer.camera.use_z_index,
        )
    _list = getattr(self, mobject_list_name)
    new_list = self.get_restructured_mobject_list(_list, to_remove)
    setattr(self, mobject_list_name, new_list)
    return self

def get_restructured_mobject_list(self, mobjects: list, to_remove: list):
    """
    Given a list of mobjects and a list of mobjects to be removed, this
    filters out the removable mobjects from the list of mobjects.

    Parameters
    -----
    mobjects
        The Mobjects to check.

    to_remove
        The list of mobjects to remove.

    Returns
    -----
    list
        The list of mobjects with the mobjects to remove removed.
    """

    new_mobjects = []

    def add_safe_mobjects_from_list(list_to_examine, set_to_remove):
        for mob in list_to_examine:
            if mob in set_to_remove:
                continue
            intersect = set_to_remove.intersection(mob.get_family())
            if intersect:
                add_safe_mobjects_from_list(mob.submobjects, intersect)
            else:
                new_mobjects.append(mob)

    add_safe_mobjects_from_list(mobjects, set(to_remove))
    return new_mobjects

# TODO, remove this, and calls to this
def add_foreground_mobjects(self, *mobjects: Mobject):
    """
    Adds mobjects to the foreground, and internally to the list
    foreground_mobjects, and mobjects.

    Parameters
    -----
    *mobjects
        The Mobjects to add to the foreground.

    Returns
    -----
    Scene
        The Scene, with the foreground mobjects added.
    """

    self.foreground_mobjects = list_update(self.foreground_mobjects, mobjects)
    self.add(*mobjects)
    return self

def add_foreground_mobject(self, mobject: Mobject):
    """
    Adds a single mobject to the foreground, and internally to the list
    foreground_mobjects, and mobjects.

    Parameters
    -----
    mobject
        The Mobject to add to the foreground.
    """

```

```

    Returns
    -----
    Scene
        The Scene, with the foreground mobject added.
    """
    return self.add_foreground_mobjects(mobject)

def remove_foreground_mobjects(self, *to_remove: Mobject):
    """
    Removes mobjects from the foreground, and internally from the list
    foreground_mobjects.

    Parameters
    -----
    *to_remove
        The mobject(s) to remove from the foreground.

    Returns
    -----
    Scene
        The Scene, with the foreground mobjects removed.
    """
    self.restructure_mobjects(to_remove, "foreground_mobjects")
    return self

def remove_foreground_mobject(self, mobject: Mobject):
    """
    Removes a single mobject from the foreground, and internally from the list
    foreground_mobjects.

    Parameters
    -----
    mobject
        The mobject to remove from the foreground.

    Returns
    -----
    Scene
        The Scene, with the foreground mobject removed.
    """
    return self.remove_foreground_mobjects(mobject)

def bring_to_front(self, *mobjects: Mobject):
    """
    Adds the passed mobjects to the scene again,
    pushing them to the front of the scene.

    Parameters
    -----
    *mobjects
        The mobject(s) to bring to the front of the scene.

    Returns
    -----
    Scene
        The Scene, with the mobjects brought to the front
        of the scene.
    """
    self.add(*mobjects)
    return self

def bring_to_back(self, *mobjects: Mobject):
    """
    Removes the mobject from the scene and
    adds them to the back of the scene.

    Parameters
    -----
    *mobjects
        The mobject(s) to push to the back of the scene.

    Returns
    -----
    Scene
        The Scene, with the mobjects pushed to the back
        of the scene.
    """
    self.remove(*mobjects)
    self.mobjects = list(mobjects) + self.mobjects
    return self

def clear(self):
    """
    Removes all mobjects present in self.mobjects
    and self.foreground_mobjects from the scene.

    Returns
    -----
    Scene
        The Scene, with all of its mobjects in
        self.mobjects and self.foreground_mobjects
        removed.
    """
    self.mobjects = []
    self.foreground_mobjects = []
    return self

def get_moving_mobjects(self, *animations: Animation):
    """
    Gets all moving mobjects in the passed animation(s).

    Parameters
    -----
    *animations
        The animations to check for moving mobjects.

    Returns
    -----
    list
        The list of mobjects that could be moving in
        the Animation(s)
    """
    # Go through mobjects from start to end, and
    # as soon as there's one that needs updating of
    # some kind per frame, return the list from that
    # point forward.
    animation_mobjects = [anim.mobject for anim in animations]
    mobjects = self.get_mobject_family_members()
    for i, mob in enumerate(mobjects):
        update_possibilities = [

```

```

        mob in animation_mobjects,
        len(mob.get_family_updaters()) > 0,
        mob in self.foreground_mobjects,
    ]
    if any(update_possibilities):
        return mobjects[i:]
    return []

def get_moving_and_static_mobjects(self, animations):
    all_mobjects = list_update(self.mobjects, self.foreground_mobjects)
    all_mobject_families = extract_mobject_family_members(
        all_mobjects,
        use_z_index=self.renderer.camera.use_z_index,
        only_those_with_points=True,
    )
    moving_mobjects = self.get_moving_mobjects(*animations)
    all_moving_mobject_families = extract_mobject_family_members(
        moving_mobjects,
        use_z_index=self.renderer.camera.use_z_index,
    )
    static_mobjects = list_difference_update(
        all_mobject_families,
        all_moving_mobject_families,
    )
    return all_moving_mobject_families, static_mobjects

def compile_animations(self, *args: Animation, **kwargs):
    """
    Creates _MethodAnimations from any _AnimationBuilders and updates animation
    kwargs with kwargs passed to play().

    Parameters
    -----
    *args
        Animations to be played.
    **kwargs
        Configuration for the call to play().

    Returns
    -----
    Tuple[:class:`Animation`]
        Animations to be played.
    """
    animations = []
    for arg in args:
        try:
            animations.append(prepare_animation(arg))
        except TypeError:
            if inspect.ismethod(arg):
                raise TypeError(
                    "Passing Mobject methods to Scene.play is no longer"
                    " supported. Use Mobject.animate instead."
                )
            else:
                raise TypeError(
                    f"Unexpected argument {arg} passed to Scene.play().",
                )

    for animation in animations:
        for k, v in kwargs.items():
            setattr(animation, k, v)

    return animations

def _get_animation_time_progression(
    self, animations: list[Animation], duration: float
):
    """
    You will hardly use this when making your own animations.
    This method is for Manim's internal use.

    Uses :func:`~.get_time_progression` to obtain a
    CommandLine ProgressBar whose ``fill_time`` is
    dependent on the qualities of the passed Animation.

    Parameters
    -----
    animations
        The list of animations to get
        the time progression for.

    duration
        duration of wait time

    Returns
    -----
    time_progression
        The CommandLine Progress Bar.
    """
    if len(animations) == 1 and isinstance(animations[0], Wait):
        stop_condition = animations[0].stop_condition
        if stop_condition is not None:
            time_progression = self.get_time_progression(
                duration,
                f"Waiting for {stop_condition.__name__}",
                n_iterations=-1, # So it doesn't show % progress
                override_skip_animations=True,
            )
        else:
            time_progression = self.get_time_progression(
                duration,
                f"Waiting {self.renderer.num_plays}",
            )
    else:
        time_progression = self.get_time_progression(
            duration,
            "".join(
                [
                    f"Animation {self.renderer.num_plays}: ",
                    str(animations[0]),
                    ("," etc." if len(animations) > 1 else ""),
                ],
            ),
        )
    return time_progression

def get_time_progression(
    self,
    run_time: float,
    description,
    n_iterations: int | None = None,

```

```

    override_skip_animations: bool = False,
):
"""
You will hardly use this when making your own animations.
This method is for Manim's internal use.

Returns a CommandLine ProgressBar whose ``fill_time`` is dependent on the ``run_time`` of an animation, the iterations to perform in that animation and a bool saying whether or not to consider the skipped animations.

Parameters
-----
run_time
    The ``run_time`` of the animation.

n_iterations
    The number of iterations in the animation.

override_skip_animations
    Whether or not to show skipped animations in the progress bar.

Returns
-----
time_progression
    The CommandLine Progress Bar.

"""
if self.renderer.skip_animations and not override_skip_animations:
    times = [run_time]
else:
    step = 1 / config["frame_rate"]
    times = np.arange(0, run_time, step)
    time_progression = tqdm(
        times,
        desc=description,
        total=n_iterations,
        leave=config["progress_bar"] == "leave",
        ascii=True if platform.system() == "Windows" else None,
        disable=config["progress_bar"] == "none",
    )
return time_progression

def get_run_time(self, animations: list[Animation]):
"""
Gets the total run time for a list of animations.

Parameters
-----
animations
    A list of the animations whose total ``run_time`` is to be calculated.

Returns
-----
float
    The total ``run_time`` of all of the animations in the list.

"""
if len(animations) == 1 and isinstance(animations[0], Wait):
    return animations[0].duration
else:
    return np.max([animation.run_time for animation in animations])

def play(
    self,
    *args,
    subcaption=None,
    subcaption_duration=None,
    subcaption_offset=0,
    **kwargs,
):
    """
    Plays an animation in this scene.

    Parameters
    -----
    args
        Animations to be played.
    subcaption
        The content of the external subcaption that should be added during the animation.
    subcaption_duration
        The duration for which the specified subcaption is added. If ``None`` (the default), the run time of the animation is taken.
    subcaption_offset
        An offset (in seconds) for the start time of the added subcaption.
    kwargs
        All other keywords are passed to the renderer.

    """
    # If we are in interactive embedded mode, make sure this is running on the main thread (required for OpenGL)
    if (
        self.interactive_mode
        and config.renderer == RendererType.OPENGL
        and threading.current_thread().name != "MainThread"
    ):
        kwargs.update(
            {
                "subcaption": subcaption,
                "subcaption_duration": subcaption_duration,
                "subcaption_offset": subcaption_offset,
            }
        )
        self.queue.put(
            (
                "play",
                args,
                kwargs,
            )
        )
    return

    start_time = self.renderer.time
    self.renderer.play(self, *args, **kwargs)
    run_time = self.renderer.time - start_time
    if subcaption:
        if subcaption_duration is None:

```

```

    subcaption_duration = run_time
    # The start of the subcaption needs to be offset by the
    # run_time of the animation because it is added after
    # the animation has already been played (and Scene.renderer.time
    # has already been updated).
    self.add_subcaption(
        content=subcaption,
        duration=subcaption.duration,
        offset=-run_time + subcaption_offset,
    )

def wait(
    self,
    duration: float = DEFAULT_WAIT_TIME,
    stop_condition: Callable[[], bool] | None = None,
    frozen_frame: bool | None = None,
):
    """Plays a "no operation" animation.

    Parameters
    -----
    duration
        The run time of the animation.
    stop_condition
        A function without positional arguments that is evaluated every time
        a frame is rendered. The animation only stops when the return value
        of the function is truthy, or when the time specified in ``duration``
        passes.
    frozen_frame
        If True, updater functions are not evaluated, and the animation outputs
        a frozen frame. If False, updater functions are called and frames
        are rendered as usual. If None (the default), the scene tries to
        determine whether or not the frame is frozen on its own.

    See also
    -----
    :class:`.Wait`, :meth:`.should_mobjects_update`  

    """
    self.play(
        Wait(
            run_time=duration,
            stop_condition=stop_condition,
            frozen_frame=frozen_frame,
        )
    )

def pause(self, duration: float = DEFAULT_WAIT_TIME):
    """Pauses the scene (i.e., displays a frozen frame).

    This is an alias for :meth:`.wait` with ``frozen_frame`` set to ``True``.

    Parameters
    -----
    duration
        The duration of the pause.

    See also
    -----
    :meth:`.wait`, :class:`.Wait`  

    """
    self.wait(duration=duration, frozen_frame=True)

def wait_until(self, stop_condition: Callable[[], bool], max_time: float = 60):
    """Wait until a condition is satisfied, up to a given maximum duration.

    Parameters
    -----
    stop_condition
        A function with no arguments that determines whether or not the
        scene should keep waiting.
    max_time
        The maximum wait time in seconds.
    """
    self.wait(max_time, stop_condition=stop_condition)

def compile_animation_data(self, *animations: Animation, **play_kwargs):
    """Given a list of animations, compile the corresponding
    static and moving mobjects, and gather the animation durations.

    This also begins the animations.

    Parameters
    -----
    animations
        Animation or mobject with mobject method and params
    play_kwargs
        Named parameters affecting what was passed in ``animations``,
        e.g. ``run_time``, ``lag_ratio`` and so on.

    Returns
    -----
    self, None
        None if there is nothing to play, or self otherwise.
    """
    # NOTE TODO : returns statement of this method are wrong. It should return nothing, as it makes a little sense to get any information from this method.
    # The return are kept to keep webgl renderer from breaking.
    if len(animations) == 0:
        raise ValueError("Called Scene.play with no animations")

    self.animations = self.compile_animations(*animations, **play_kwargs)
    self.add_mobjects_from_animations(self.animations)

    self.last_t = 0
    self.stop_condition = None
    self.moving_mobjects = []
    self.static_mobjects = []

    if len(self.animations) == 1 and isinstance(self.animations[0], Wait):
        if self.should_update_mobjects():
            self.update_mobjects(dt=0) # Any problems with this?
            self.stop_condition = self.animations[0].stop_condition
        else:
            self.duration = self.animations[0].duration
            # Static image logic when the wait is static is done by the renderer, not here.
            self.animations[0].is_static_wait = True
            return None
    self.duration = self.get_run_time(self.animations)
    return self

def begin_animations(self) -> None:

```

```

"""Start the animations of the scene."""
for animation in self.animations:
    animation._setup_scene(self)
    animation.begin()

if config.renderer == RendererType.CAIRO:
    # Paint all non-moving objects onto the screen, so they don't
    # have to be rendered every frame
    (
        self.moving_mobjects,
        self.static_mobjects,
    ) = self.get_moving_and_static_mobjects(self.animations)

def is_current_animation_frozen_frame(self) -> bool:
    """Returns whether the current animation produces a static frame (generally a Wait)."""
    return (
        isinstance(self.animations[0], Wait)
        and len(self.animations) == 1
        and self.animations[0].is_static_wait
    )

def play_internal(self, skip_rendering: bool = False):
    """
    This method is used to prep the animations for rendering,
    apply the arguments and parameters required to them,
    render them, and write them to the video file.

    Parameters
    -----
    skip_rendering
        Whether the rendering should be skipped, by default False
    """
    self.duration = self.get_run_time(self.animations)
    self.time_progression = self._get_animation_time_progression(
        self.animations,
        self.duration,
    )
    for t in self.time_progression:
        self.update_to_time(t)
        if not skip_rendering and not self.skip_animation_preview:
            self.renderer.render(self, t, self.moving_mobjects)
        if self.stop_condition is not None and self.stop_condition():
            self.time_progression.close()
            break

    for animation in self.animations:
        animation.finish()
        animation.clean_up_from_scene(self)
    if not self.renderer.skip_animations:
        self.update_mobjects(0)
    self.renderer.static_image = None
    # Closing the progress bar at the end of the play.
    self.time_progression.close()

def check_interactive_embed_is_valid(self):
    if config["force_window"]:
        return True
    if self.skip_animation_preview:
        logger.warning(
            "Disabling interactive embed as 'skip_animation_preview' is enabled",
        )
        return False
    elif config["write_to_movie"]:
        logger.warning("Disabling interactive embed as 'write_to_movie' is enabled")
        return False
    elif config["format"]:
        logger.warning(
            "Disabling interactive embed as '--format' is set as "
            + config["format"],
        )
        return False
    elif not self.renderer.window:
        logger.warning("Disabling interactive embed as no window was created")
        return False
    elif config.dry_run:
        logger.warning("Disabling interactive embed as dry_run is enabled")
        return False
    return True

def interactive_embed(self):
    """
    Like embed(), but allows for screen interaction.
    """
    if not self.check_interactive_embed_is_valid():
        return
    self.interactive_mode = True

def ipython(shell, namespace):
    import manim.opengl

    def load_module_into_namespace(module, namespace):
        for name in dir(module):
            namespace[name] = getattr(module, name)

    load_module_into_namespace(manim, namespace)
    load_module_into_namespace(manim.opengl, namespace)

    def embedded_rerun(*args, **kwargs):
        self.queue.put(("rerun_keyboard", args, kwargs))
        shell.exit()

    namespace["rerun"] = embedded_rerun

    shell(local_ns=namespace)
    self.queue.put(("exit_keyboard", [], {}))

def get_embedded_method(method_name):
    return lambda *args, **kwargs: self.queue.put((method_name, args, kwargs))

local_namespace = inspect.currentframe().f_back.f_locals
for method in ("play", "wait", "add", "remove"):
    embedded_method = get_embedded_method(method)
    # Allow for calling scene methods without prepending 'self.'.
    local_namespace[method] = embedded_method

from IPython.terminal.embed import InteractiveShellEmbed
from traitlets.config import Config

cfg = Config()
cfg.TerminalInteractiveShell.confirm_exit = False
shell = InteractiveShellEmbed(config=cfg)

```

```

keyboard_thread = threading.Thread(
    target=ipython,
    args=(shell, local_namespace),
)
# run as daemon to kill thread when main thread exits
if not shell.pt_app:
    keyboard_thread.daemon = True
keyboard_thread.start()

if self.dearpygui_imported and config["enable_gui"]:
    if not dpg.is_dearpygui_running():
        gui_thread = threading.Thread(
            target=configure_pygui,
            args=(self.renderer, self.widgets),
            kwargs={"update": False},
        )
        gui_thread.start()
    else:
        configure_pygui(self.renderer, self.widgets, update=True)

self.camera.model_matrix = self.camera.default_model_matrix

self.interact(shell, keyboard_thread)

def interact(self, shell, keyboard_thread):
    event_handler = RerunSceneHandler(self.queue)
    file_observer = Observer()
    file_observer.schedule(event_handler, config["input_file"], recursive=True)
    file_observer.start()

    self.quit_interaction = False
    keyboard_thread_needs_join = shell.pt_app is not None
    assert self.queue.qsize() == 0

    last_time = time.time()
    while not (self.renderer.window.is_closing or self.quit_interaction):
        if not self.queue.empty():
            tup = self.queue.get_nowait()
            if tup[0].startswith("rerun"):
                # Intentionally skip calling join() on the file thread to save time.
                if not tup[0].endswith("keyboard"):
                    if shell.pt_app:
                        shell.pt_app.app.exit(exception=EOFError)
                    file_observer.unschedule_all()
                    raise RerunSceneException
            keyboard_thread.join()

            kwargs = tup[2]
            if "from_animation_number" in kwargs:
                config["from_animation_number"] = kwargs[
                    "from_animation_number"
                ]
            # # TODO: This option only makes sense if interactive_embed() is run at the
            # # end of a scene by default.
            # if "upto_animation_number" in kwargs:
            #     config["upto_animation_number"] = kwargs[
            #         "upto_animation_number"
            #     ]

            keyboard_thread.join()
            file_observer.unschedule_all()
            raise RerunSceneException
        elif tup[0].startswith("exit"):
            # Intentionally skip calling join() on the file thread to save time.
            if not tup[0].endswith("keyboard") and shell.pt_app:
                shell.pt_app.app.exit(exception=EOFError)
            keyboard_thread.join()
            # Remove exit_keyboard from the queue if necessary.
            while self.queue.qsize() > 0:
                self.queue.get()
            keyboard_thread_needs_join = False
            break
        else:
            method, args, kwargs = tup
            getattr(self, method)(*args, **kwargs)
        else:
            self.renderer.animation_start_time = 0
            dt = time.time() - last_time
            last_time = time.time()
            self.renderer.render(self, dt, self.moving_mobjects)
            self.update_mobjects(dt)
            self.update_meshes(dt)
            self.update_self(dt)

    # Join the keyboard thread if necessary.
    if shell is not None and keyboard_thread_needs_join:
        shell.pt_app.app.exit(exception=EOFError)
        keyboard_thread.join()
        # Remove exit_keyboard from the queue if necessary.
        while self.queue.qsize() > 0:
            self.queue.get()

file_observer.stop()
file_observer.join()

if self.dearpygui_imported and config["enable_gui"]:
    dpg.stop_dearpygui()

if self.renderer.window.is_closing:
    self.renderer.window.destroy()

def embed(self):
    if not config["preview"]:
        logger.warning("Called embed() while no preview window is available.")
        return
    if config["write_to_movie"]:
        logger.warning("embed() is skipped while writing to a file.")
        return

    self.renderer.animation_start_time = 0
    self.renderer.render(self, -1, self.moving_mobjects)

    # Configure IPython shell.
    from IPython.terminal.embed import InteractiveShellEmbed

    shell = InteractiveShellEmbed()

    # Have the frame update after each command
    shell.events.register(
        "post_run_cell",

```

```

        lambda *a, **kw: self.renderer.render(self, -1, self.moving_mobjects),
    )

# Use the locals of the caller as the local namespace
# once embedded, and add a few custom shortcuts.
local_ns = inspect.currentframe().f_back.f_locals
# local_ns["touch"] = self.interact
for method in (
    "play",
    "wait",
    "add",
    "remove",
    "interact",
    # "clear",
    # "save_state",
    # "restore",
):
    local_ns[method] = getattr(self, method)
shell(local_ns=local_ns, stack_depth=2)

# End scene when exiting an embed.
raise Exception("Exiting scene.")

def update_to_time(self, t):
    dt = t - self.last_t
    self.last_t = t
    for animation in self.animations:
        animation.update_mobjects(dt)
        alpha = t / animation.run_time
        animation.interpolate(alpha)
    self.update_mobjects(dt)
    self.update_meshes(dt)
    self.update_self(dt)

def add_subcaption(
    self, content: str, duration: float = 1, offset: float = 0
) -> None:
    """Adds an entry in the corresponding subcaption file
    at the current time stamp.

    The current time stamp is obtained from ``Scene.renderer.time``.

    Parameters
    -----
    content
        The subcaption content.
    duration
        The duration (in seconds) for which the subcaption is shown.
    offset
        This offset (in seconds) is added to the starting time stamp
        of the subcaption.

    Examples
    -----
    This example illustrates both possibilities for adding
    subcaptions to Manim:::

    class SubCaptionExample(Scene):
        def construct(self):
            square = Square()
            circle = Circle()

            # first option: via the add_subcaption method
            self.add_subcaption("Hello square!", duration=1)
            self.play(Create(square))

            # second option: within the call to Scene.play
            self.play(
                Transform(square, circle),
                subcaption="The square transforms."
            )

        """
        subtitle = srt.Subtitle(
            index=len(self.renderer.file_writer.subcaptions),
            content=content,
            start=datetime.timedelta(seconds=self.renderer.time + offset),
            end=datetime.timedelta(seconds=self.renderer.time + offset + duration),
        )
        self.renderer.file_writer.subcaptions.append(subtitle)

    def add_sound(
        self,
        sound_file: str,
        time_offset: float = 0,
        gain: float | None = None,
        **kwargs,
    ):
        """
        This method is used to add a sound to the animation.

        Parameters
        -----
        sound_file
            The path to the sound file.
        time_offset
            The offset in the sound file after which
            the sound can be played.
        gain
            Amplification of the sound.

        Examples
        -----
        .. manim:: SoundExample
           :no_autoplay:

        class SoundExample(Scene):
            # Source of sound under Creative Commons 0 License. https://freesound.org/people/Druminfected/sounds/250551/
            def construct(self):
                dot = Dot().set_color(GREEN)
                self.add_sound("click.wav")
                self.add(dot)
                self.wait()
                self.add_sound("click.wav")
                dot.set_color(BLUE)
                self.wait()
                self.add_sound("click.wav")
                dot.set_color(RED)

```

```

    self.wait()

Download the resource for the previous example `here <https://github.com/ManimCommunity/manim/blob/main/docs/source/\_static/click.wav>`_.

"""
if self.renderer.skip_animations:
    return
time = self.renderer.time + time_offset
self.renderer.file_writer.add_sound(sound_file, time, gain, **kwargs)

def on_mouse_motion(self, point, d_point):
    self.mouse_point.move_to(point)
    if SHIFT_VALUE in self.renderer.pressed_keys:
        shift = -d_point
        shift[0] *= self.camera.get_width() / 2
        shift[1] *= self.camera.get_height() / 2
        transform = self.camera.inverse_rotation_matrix
        shift = np.dot(np.transpose(transform), shift)
        self.camera.shift(shift)

def on_mouse_scroll(self, point, offset):
    if not config.use_projection_stroke_shaders:
        factor = 1 + np.arctan(-2.1 * offset[1])
        self.camera.scale(factor, about_point=self.camera_target)
    self.mouse_scroll_orbit_controls(point, offset)

def on_key_press(self, symbol, modifiers):
    try:
        char = chr(symbol)
    except OverflowError:
        logger.warning("The value of the pressed key is too large.")
        return

    if char == "r":
        self.camera.to_default_state()
        self.camera_target = np.array([0, 0, 0], dtype=np.float32)
    elif char == "q":
        self.quit_interaction = True
    else:
        if char in self.key_to_function_map:
            self.key_to_function_map[char]()

def on_key_release(self, symbol, modifiers):
    pass

def on_mouse_drag(self, point, d_point, buttons, modifiers):
    self.mouse_drag_point.move_to(point)
    if buttons == 1:
        self.camera.increment_theta(-d_point[0])
        self.camera.increment_phi(d_point[1])
    elif buttons == 4:
        camera_x_axis = self.camera.model_matrix[:, 0]
        horizontal_shift_vector = -d_point[0] * camera_x_axis
        vertical_shift_vector = -d_point[1] * np.cross(OUP, camera_x_axis)
        total_shift_vector = horizontal_shift_vector + vertical_shift_vector
        self.camera.shift(1.1 * total_shift_vector)

    self.mouse_drag_orbit_controls(point, d_point, buttons, modifiers)

def mouse_scroll_orbit_controls(self, point, offset):
    camera_to_target = self.camera_target - self.camera.get_position()
    camera_to_target *= np.sign(offset[1])
    shift_vector = 0.01 * camera_to_target
    self.camera.model_matrix = (
        opengl.translation_matrix(*shift_vector) @ self.camera.model_matrix
    )

def mouse_drag_orbit_controls(self, point, d_point, buttons, modifiers):
    # Left click drag.
    if buttons == 1:
        # Translate to target the origin and rotate around the z axis.
        self.camera.model_matrix = (
            opengl.rotation_matrix(z=-d_point[0])
            @ opengl.translation_matrix(*-self.camera_target)
            @ self.camera.model_matrix
        )

        # Rotation off of the z axis.
        camera_position = self.camera.get_position()
        camera_y_axis = self.camera.model_matrix[:, 1]
        axis_of_rotation = space_ops.normalize(
            np.cross(camera_y_axis, camera_position),
        )
        rotation_matrix = space_ops.rotation_matrix(
            d_point[1],
            axis_of_rotation,
            homogeneous=True,
        )

        maximum_polar_angle = self.camera.maximum_polar_angle
        minimum_polar_angle = self.camera.minimum_polar_angle

        potential_camera_model_matrix = rotation_matrix @ self.camera.model_matrix
        potential_camera_location = potential_camera_model_matrix[:, 0]
        potential_camera_y_axis = potential_camera_model_matrix[:, 1]
        sign = (
            np.sign(potential_camera_y_axis[2])
            if potential_camera_y_axis[2] != 0
            else 1
        )
        potential_polar_angle = sign * np.arccos(
            potential_camera_location[2]
            / np.linalg.norm(potential_camera_location),
        )
        if minimum_polar_angle <= potential_polar_angle <= maximum_polar_angle:
            self.camera.model_matrix = potential_camera_model_matrix
        else:
            sign = np.sign(camera_y_axis[2]) if camera_y_axis[2] != 0 else 1
            current_polar_angle = sign * np.arccos(
                camera_position[2] / np.linalg.norm(camera_position),
            )
            if potential_polar_angle > maximum_polar_angle:
                polar_angle_delta = maximum_polar_angle - current_polar_angle
            else:
                polar_angle_delta = minimum_polar_angle - current_polar_angle
            rotation_matrix = space_ops.rotation_matrix(
                polar_angle_delta,
                axis_of_rotation,
                homogeneous=True,
            )
            self.camera.model_matrix = rotation_matrix @ self.camera.model_matrix

```

```
# Translate to target the original target.
self.camera.model_matrix = (
    opengl.translation_matrix(*self.camera_target)
    @ self.camera.model_matrix
)
# Right click drag.
elif buttons == 4:
    camera_x_axis = self.camera.model_matrix[:3, 0]
    horizontal_shift_vector = -d_point[0] * camera_x_axis
    vertical_shift_vector = -d_point[1] * np.cross(OUp, camera_x_axis)
    total_shift_vector = horizontal_shift_vector + vertical_shift_vector

    self.camera.model_matrix = (
        opengl.translation_matrix(*total_shift_vector)
        @ self.camera.model_matrix
)
    self.camera_target += total_shift_vector

def set_key_function(self, char, func):
    self.key_to_function_map[char] = func

def on_mouse_press(self, point, button, modifiers):
    for func in self.mouse_press_callbacks:
        func()
```

composition.py

```
"""Tools for displaying multiple animations at once."""

from __future__ import annotations

from typing import TYPE_CHECKING, Callable, Sequence

import numpy as np

from manim.mobject.opengl.opengl_mobject import OpenGLGroup

from ..config import config
from ..animation.animation import Animation, prepare_animation
from ..constants import RendererType
from ..mobject.mobject import Group, Mobject
from ..scene.scene import Scene
from ..utils.iterables import remove_list_redundancies
from ..utils.rate_functions import linear

if TYPE_CHECKING:
    from manim.mobject.opengl_vectorized_mobject import OpenGLVGroup
    from ..mobject.types.vectorized_mobject import VGroup

__all__ = ["AnimationGroup", "Succession", "LaggedStart", "LaggedStartMap"]

DEFAULT_LAGGED_START_LAG_RATIO: float = 0.05

class AnimationGroup(Animation):
    """Plays a group or series of :class:`~.Animation`.

    Parameters
    -----
    animations
        Sequence of :class:`~.Animation` objects to be played.
    group
        A group of multiple :class:`~.Mobject`.
    run_time
        The duration of the animation in seconds.
    rate_func
        The function defining the animation progress based on the relative
        runtime (see :mod:`~.rate_functions`).
    lag_ratio
        Defines the delay after which the animation is applied to submobjects. A lag_ratio of
        ``n.nn`` means the next animation will play when ``nnn%`` of the current animation has played.
        Defaults to 0.0, meaning that all animations will be played together.

        This does not influence the total runtime of the animation. Instead the runtime
        of individual animations is adjusted so that the complete animation has the defined
        run time.
    """

    def __init__(
        self,
        *animations: Animation,
        group: Group | VGroup | OpenGLGroup | OpenGLVGroup = None,
        run_time: float | None = None,
        rate_func: Callable[[float], float] = linear,
        lag_ratio: float = 0,
        **kwargs,
    ) -> None:
        self.animations = [prepare_animation(anim) for anim in animations]
        self.rate_func = rate_func
        self.group = group
        if self.group is None:
            mobjects = remove_list_redundancies(
                [anim.mobject for anim in self.animations if not anim.is_introducer()],
            )
            if config["renderer"] == RendererType.OPENGL:
                self.group = OpenGLGroup(*mobjects)
            else:
                self.group = Group(*mobjects)
        super().__init__(
            self.group, rate_func=self.rate_func, lag_ratio=lag_ratio, **kwargs
        )
        self.run_time: float = self.init_run_time(run_time)

    def get_all_mobjects(self) -> Sequence[Mobject]:
        return list(self.group)
```

```

def begin(self) -> None:
    if self.suspend_mobject_updating:
        self.group.suspend_updating()
    for anim in self.animations:
        anim.begin()

def _setup_scene(self, scene) -> None:
    for anim in self.animations:
        anim._setup_scene(scene)

def finish(self) -> None:
    for anim in self.animations:
        anim.finish()
    if self.suspend_mobject_updating:
        self.group.resume_updating()

def clean_up_from_scene(self, scene: Scene) -> None:
    self._on_finish(scene)
    for anim in self.animations:
        if self.remover:
            anim.remover = self.remover
    anim.clean_up_from_scene(scene)

def update_mobjects(self, dt: float) -> None:
    for anim in self.animations:
        anim.update_mobjects(dt)

def init_run_time(self, run_time) -> float:
    """Calculates the run time of the animation, if different from ``run_time``.

    Parameters
    -----
    run_time
        The duration of the animation in seconds.

    Returns
    -----
    run_time
        The duration of the animation in seconds.
    """
    self.build_animations_with_timings()
    if self.anims_with_timings:
        self.max_end_time = np.max([awt[2] for awt in self.anims_with_timings])
    else:
        self.max_end_time = 0
    return self.max_end_time if run_time is None else run_time

def build_animations_with_timings(self) -> None:
    """Creates a list of triplets of the form (anim, start_time, end_time)."""
    self.anims_with_timings = []
    curr_time: float = 0
    for anim in self.animations:
        start_time: float = curr_time
        end_time: float = start_time + anim.get_run_time()
        self.anims_with_timings.append((anim, start_time, end_time))
        # Start time of next animation is based on the lag_ratio
        curr_time = (1 - self.lag_ratio) * start_time + self.lag_ratio * end_time

def interpolate(self, alpha: float) -> None:
    # Note, if the run_time of AnimationGroup has been
    # set to something other than its default, these
    # times might not correspond to actual times,
    # e.g. of the surrounding scene. Instead they'd
    # be a rescaled version. But that's okay!
    time = self.rate_func(alpha) * self.max_end_time
    for anim, start_time, end_time in self.anims_with_timings:
        anim_time = end_time - start_time
        if anim_time == 0:
            sub_alpha = 0
        else:
            sub_alpha = np.clip((time - start_time) / anim_time, 0, 1)
        anim.interpolate(sub_alpha)

class Succession(AnimationGroup):
    """Plays a series of animations in succession.

    Parameters
    -----
    animations
        Sequence of :class:`~.Animation` objects to be played.
    lag_ratio
        Defines the delay after which the animation is applied to subobjects. A lag_ratio of
        ``n.nn`` means the next animation will play when ``nnn%`` of the current animation has played.
        Defaults to 1.0, meaning that the next animation will begin when 100% of the current
        animation has played.
    """

```

This does not influence the total runtime of the animation. Instead the runtime of individual animations is adjusted so that the complete animation has the defined run time.

Examples

```
.. manim:: SuccessionExample

    class SuccessionExample(Scene):
        def construct(self):
            dot1 = Dot(point=LEFT * 2 + UP * 2, radius=0.16, color=BLUE)
            dot2 = Dot(point=LEFT * 2 + DOWN * 2, radius=0.16, color=MAROON)
            dot3 = Dot(point=RIGHT * 2 + DOWN * 2, radius=0.16, color=GREEN)
            dot4 = Dot(point=RIGHT * 2 + UP * 2, radius=0.16, color=YELLOW)
            self.add(dot1, dot2, dot3, dot4)

            self.play(Succession(
                dot1.animate.move_to(dot2),
                dot2.animate.move_to(dot3),
                dot3.animate.move_to(dot4),
                dot4.animate.move_to(dot1)
            ))
        """

    def __init__(self, *animations: Animation, lag_ratio: float = 1, **kwargs) -> None:
        super().__init__(*animations, lag_ratio=lag_ratio, **kwargs)

    def begin(self) -> None:
        assert len(self.animations) > 0
        self.update_active_animation(0)

    def finish(self) -> None:
        while self.active_animation is not None:
            self.next_animation()

    def update_mobjects(self, dt: float) -> None:
        if self.active_animation:
            self.active_animation.update_mobjects(dt)

    def _setup_scene(self, scene) -> None:
        if scene is None:
            return
        if self.is_introducer():
            for anim in self.animations:
                if not anim.is_introducer() and anim.mobject is not None:
                    scene.add(anim.mobject)

        self.scene = scene

    def update_active_animation(self, index: int) -> None:
        self.active_index = index
        if index >= len(self.animations):
            self.active_animation: Animation | None = None
            self.active_start_time: float | None = None
            self.active_end_time: float | None = None
        else:
            self.active_animation = self.animations[index]
            self.active_animation._setup_scene(self.scene)
            self.active_animation.begin()
            self.active_start_time = self.anims_with_timings[index][1]
            self.active_end_time = self.anims_with_timings[index][2]

    def next_animation(self) -> None:
        """Proceeds to the next animation.

        This method is called right when the active animation finishes.
        """
        if self.active_animation is not None:
            self.active_animation.finish()
        self.update_active_animation(self.active_index + 1)

    def interpolate(self, alpha: float) -> None:
        current_time = self.rate_func(alpha) * self.max_end_time
        while self.active_end_time is not None and current_time >= self.active_end_time:
            self.next_animation()
        if self.active_animation is not None and self.active_start_time is not None:
            elapsed = current_time - self.active_start_time
            active_run_time = self.active_animation.get_run_time()
            subalpha = elapsed / active_run_time if active_run_time != 0.0 else 1.0
            self.active_animation.interpolate(subalpha)

    class LaggedStart(AnimationGroup):
        """Adjusts the timing of a series of :class:`~.Animation` according to ``lag_ratio``.

        Parameters

```

```

-----
animations
    Sequence of :class:`~.Animation` objects to be played.
lag_ratio
    Defines the delay after which the animation is applied to submobjects. A lag_ratio of
    ``n.nn`` means the next animation will play when ``nnn%`` of the current animation has played.
    Defaults to 0.05, meaning that the next animation will begin when 5% of the current
    animation has played.

    This does not influence the total runtime of the animation. Instead the runtime
    of individual animations is adjusted so that the complete animation has the defined
    run time.

Examples
-----
.. manim:: LaggedStartExample

    class LaggedStartExample(Scene):
        def construct(self):
            title = Text("lag_ratio = 0.25").to_edge(UP)

            dot1 = Dot(point=LEFT * 2 + UP, radius=0.16)
            dot2 = Dot(point=LEFT * 2, radius=0.16)
            dot3 = Dot(point=LEFT * 2 + DOWN, radius=0.16)
            line_25 = DashedLine(
                start=LEFT + UP * 2,
                end=LEFT + DOWN * 2,
                color=RED
            )
            label = Text("25%", font_size=24).next_to(line_25, UP)
            self.add(title, dot1, dot2, dot3, line_25, label)

            self.play(LaggedStart(
                dot1.animate.shift(RIGHT * 4),
                dot2.animate.shift(RIGHT * 4),
                dot3.animate.shift(RIGHT * 4),
                lag_ratio=0.25,
                run_time=4
            ))
    """
def __init__(
    self,
    *animations: Animation,
    lag_ratio: float = DEFAULT_LAGGED_START_LAG_RATIO,
    **kwargs,
):
    super().__init__(*animations, lag_ratio=lag_ratio, **kwargs)

class LaggedStartMap(LaggedStart):
    """Plays a series of :class:`~.Animation` while mapping a function to submobjects.

Parameters
-----
AnimationClass
    :class:`~.Animation` to apply to mobject.
mobject
    :class:`~.Mobject` whose submobjects the animation, and optionally the function,
    are to be applied.
arg_creator
    Function which will be applied to :class:`~.Mobject`.
run_time
    The duration of the animation in seconds.

Examples
-----
.. manim:: LaggedStartMapExample

    class LaggedStartMapExample(Scene):
        def construct(self):
            title = Tex("LaggedStartMap").to_edge(UP, buff=LARGE_BUFF)
            dots = VGroup(
                *[Dot(radius=0.16) for _ in range(35)]
            ).arrange_in_grid(rows=5, cols=7, buff=MED_LARGE_BUFF)
            self.add(dots, title)

            # Animate yellow ripple effect
            for mob in dots, title:
                self.play(LaggedStartMap(
                    ApplyMethod, mob,
                    lambda m : (m.set_color, YELLOW),
                    lag_ratio = 0.1,
                    rate_func = there_and_back,
                    run_time = 2
                ))
    """

```

```
def __init__(  
    self,  
    AnimationClass: Callable[..., Animation],  
    mobject: Mobject,  
    arg_creator: Callable[[Mobject], str] = None,  
    run_time: float = 2,  
    **kwargs,  
) -> None:  
    args_list = []  
    for submob in mobject:  
        if arg_creator:  
            args_list.append(arg_creator(submob))  
        else:  
            args_list.append((submob,))  
    anim_kwargs = dict(kwargs)  
    if "lag_ratio" in anim_kwargs:  
        anim_kwargs.pop("lag_ratio")  
    animations = [AnimationClass(*args, **anim_kwargs) for args in args_list]  
    super().__init__(*animations, run_time=run_time, **kwargs)
```

text_mobject.py

```
"""Mobjects used for displaying (non-LaTeX) text.

.. note::
    Just as you can use :class:`~.Tex` and :class:`~.MathTex` (from the module :mod:`~.tex_mobject`)
    to insert LaTeX to your videos, you can use :class:`~.Text` to add normal text.

.. important::

    See the corresponding tutorial :ref:`using-text-objects`, especially for information about fonts.

The simplest way to add text to your animations is to use the :class:`~.Text` class. It uses the Pango library to render text.
With Pango, you are also able to render non-English alphabets like 'àéàé%' or 'â"â,"â«â;â' or 'i•^é...•i•"i,,iš"' or 'Ü...Ø±ØØ"Ø Ø"ØSÙ„Ø¹ØSÙ„Ù...'.

Examples
-----
.. manim:: HelloWorld
    :save_last_frame:

    class HelloWorld(Scene):
        def construct(self):
            text = Text('Hello world').scale(3)
            self.add(text)

.. manim:: TextAlignment
    :save_last_frame:

    class TextAlignment(Scene):
        def construct(self):
            title = Text("K-means clustering and Logistic Regression", color=WHITE)
            title.scale(0.75)
            self.add(title.to_edge(UP))

            t1 = Text("1. Measuring").set_color(WHITE)
            t2 = Text("2. Clustering").set_color(WHITE)
            t3 = Text("3. Regression").set_color(WHITE)
            t4 = Text("4. Prediction").set_color(WHITE)

            x = VGroup(t1, t2, t3, t4).arrange(direction=DOWN, aligned_edge=LEFT).scale(0.7).next_to(ORIGIN, DR)
            x.set_opacity(0.5)
            x.submobjects[1].set_opacity(1)
            self.add(x)

"""

from __future__ import annotations

__all__ = ["Text", "Paragraph", "MarkupText", "register_font"]

import copy
import hashlib
import os
import re
from contextlib import contextmanager
from itertools import chain
from pathlib import Path
from typing import Iterable, Sequence

import manimpango
import numpy as np
from colour import Color
from manimpango import MarkupUtils, PangoUtils, TextSetting

from manim import config, logger
from manim.constants import *
from manim.mobject.geometry.arc import Dot
from manim.mobject.svg.svg_mobject import SVGMOBObject
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.color import Colors, color_gradient
from manim.utils.deprecation import deprecated

TEXT_MOB_SCALE_FACTOR = 0.05
DEFAULT_LINE_SPACING_SCALE = 0.3
TEXT2SVG_ADJUSTMENT_FACTOR = 4.8

def remove_invisible_chars(mobject: SVGMOBObject) -> SVGMOBObject:
    """Function to remove unwanted invisible characters from some mobjects.

    Parameters
    -----
    mobject
        Any SVGMOBObject from which we want to remove unwanted invisible characters.

    Returns
    -----
    :class:`~.SVGMOBObject`
        The SVGMOBObject without unwanted invisible characters.
    """
    # TODO: Refactor needed
    iscode = False
    if mobject.__class__.__name__ == "Text":
        mobject = mobject[:]
    elif mobject.__class__.__name__ == "Code":
        iscode = True
        code = mobject
        mobject = mobject.code
    mobject_without_dots = VGroup()
    if mobject[0].__class__ == VGroup:
        for i in range(len(mobject)):
            mobject_without_dots.add(VGroup())
            mobject_without_dots[i].add(*[k for k in mobject[i] if k.__class__ != Dot])
    else:
```

```

mobject_without_dots.add(*[k for k in mobject if k.__class__ != Dot])
if iscode:
    code.code = mobject_without_dots
    return code
return mobject_without_dots

class Paragraph(VGroup):
    """Display a paragraph of text.

    For a given :class:`.Paragraph` ``par``, the attribute ``par.chars`` is a
    :class:`.VGroup` containing all the lines. In this context, every line is
    constructed as a :class:`.VGroup` of characters contained in the line.

    Parameters
    -----
    line_spacing
        Represents the spacing between lines. Defaults to -1, which means auto.
    alignment
        Defines the alignment of paragraph. Defaults to None. Possible values are "left", "right" or "center".
    Examples
    -----
    Normal usage::

        paragraph = Paragraph('this is a awesome', 'paragraph',
            'With \nNewlines', '\tWith Tabs',
            ' With Spaces', 'With Alignments',
            'center', 'left', 'right')

    Remove unwanted invisible characters::

        self.play(Transform(remove_invisible_chars(paragraph.chars[0:2]),
            remove_invisible_chars(paragraph.chars[3][0:3])))

    """
    def __init__(self,
                 *text: Sequence[str],
                 line_spacing: float = -1,
                 alignment: Optional[str] = None,
                 **kwargs,
                 ) -> None:
        self.line_spacing = line_spacing
        self.alignment = alignment
        self.consider_spaces_as_chars = kwargs.get("disable_ligatures", False)
        super().__init__()

        lines_str = "\n".join(list(text))
        self.lines_text = Text(lines_str, line_spacing=line_spacing, **kwargs)
        lines_str_list = lines_str.split("\n")
        self.chars = self._gen_chars(lines_str_list)

        self.lines = [list(self.chars), [self.alignment] * len(self.chars)]
        self.lines_initial_positions = [line.get_center() for line in self.lines[0]]
        self.add(self.lines[0])
        self.move_to(np.array([0, 0, 0]))
        if self.alignment:
            self._set_all_lines_alignments(self.alignment)

    def _gen_chars(self, lines_str_list: list) -> VGroup:
        """Function to convert a list of plain strings to a VGroup of VGroups of chars.

        Parameters
        -----
        lines_str_list
            List of plain text strings.

        Returns
        -----
        :class:`.~.VGroup`
            The generated 2d-VGroup of chars.
        """
        char_index_counter = 0
        chars = self.get_group_class()()
        for line_no in range(len(lines_str_list)):
            line_str = lines_str_list[line_no]
            # Count all the characters in line_str
            # Spaces may or may not count as characters
            if self.consider_spaces_as_chars:
                char_count = len(line_str)
            else:
                char_count = 0
                for char in line_str:
                    if not char.isspace():
                        char_count += 1

            chars.add(self.get_group_class()())
            chars[line_no].add(
                *self.lines_text.chars[
                    char_index_counter : char_index_counter + char_count
                ]
            )
            char_index_counter += char_count
            if self.consider_spaces_as_chars:
                # If spaces count as characters, count the extra \n character
                # which separates Paragraph's lines to avoid issues
                char_index_counter += 1
        return chars

    def _set_all_lines_alignments(self, alignment: str) -> Paragraph:
        """Function to set all line's alignment to a specific value.

        Parameters
        -----
        alignment
            Defines the alignment of paragraph. Possible values are "left", "right", "center".
        """
        for line_no in range(len(self.lines[0])):
            self._change_alignment_for_a_line(alignment, line_no)
        return self

```

```

def _set_line_alignment(self, alignment: str, line_no: int) -> Paragraph:
    """Function to set one line's alignment to a specific value.

    Parameters
    -----
    alignment
        Defines the alignment of paragraph. Possible values are "left", "right", "center".
    line_no
        Defines the line number for which we want to set given alignment.
    """
    self._change_alignment_for_a_line(alignment, line_no)
    return self

def _set_all_lines_to_initial_positions(self) -> Paragraph:
    """Set all lines to their initial positions."""
    self.lines[1] = [None] * len(self.lines[0])
    for line_no in range(len(self.lines[0])):
        self[line_no].move_to(
            self.get_center() + self.lines_initial_positions[line_no],
        )
    return self

def _set_line_to_initial_position(self, line_no: int) -> Paragraph:
    """Function to set one line to initial positions.

    Parameters
    -----
    line_no
        Defines the line number for which we want to set given alignment.
    """
    self.lines[1][line_no] = None
    self[line_no].move_to(self.get_center() + self.lines_initial_positions[line_no])
    return self

def _change_alignment_for_a_line(self, alignment: str, line_no: int) -> None:
    """Function to change one line's alignment to a specific value.

    Parameters
    -----
    alignment
        Defines the alignment of paragraph. Possible values are "left", "right", "center".
    line_no
        Defines the line number for which we want to set given alignment.
    """
    self.lines[1][line_no] = alignment
    if self.lines[1][line_no] == "center":
        self[line_no].move_to(
            np.array([self.get_center()[0], self[line_no].get_center()[1], 0]),
        )
    elif self.lines[1][line_no] == "right":
        self[line_no].move_to(
            np.array([
                self.get_right()[0] - self[line_no].width / 2,
                self[line_no].get_center()[1],
                0,
            ]),
        )
    elif self.lines[1][line_no] == "left":
        self[line_no].move_to(
            np.array([
                self.get_left()[0] + self[line_no].width / 2,
                self[line_no].get_center()[1],
                0,
            ]),
        )
    )

class Text(SVGMobject):
    """Display (non-LaTeX) text rendered using `Pango <https://pango.gnome.org/>`_.

    Text objects behave like a :class:`.VGroup`-like iterable of all characters in the given text. In particular, slicing is possible.

    Parameters
    -----
    text
        The text that needs to be created as a mobject.
    font
        The font family to be used to render the text. This is either a system font or one loaded with `register_font()`. Note that font family names may be different across operating systems.
    warn_missing_font
        If True (default), Manim will issue a warning if the font does not exist in the (case-sensitive) list of fonts returned from `manimpango.list_fonts()`.

    Returns
    -----
    :class:`Text`
        The mobject-like :class:`.VGroup`.

    Examples
    -----
    .. manim:: Example1Text
       :save_last_frame:

       class Example1Text(Scene):
           def construct(self):
               text = Text('Hello world').scale(3)
               self.add(text)

    .. manim:: TextColorExample
       :save_last_frame:

       class TextColorExample(Scene):
           def construct(self):
               text1 = Text('Hello world', color=BLUE).scale(3)
               text2 = Text('Hello world', gradient=(BLUE, GREEN)).scale(3).next_to(text1, DOWN)
               self.add(text1, text2)

```



```

t2c = {}
if t2f is None:
    t2f = {}
if t2g is None:
    t2g = {}
if t2s is None:
    t2s = {}
if t2w is None:
    t2w = {}
# If long form arguments are present, they take precedence
t2c = kargs.pop("text2color", t2c)
t2f = kargs.pop("text2font", t2f)
t2g = kargs.pop("text2gradient", t2g)
t2s = kargs.pop("text2slant", t2s)
t2w = kargs.pop("text2weight", t2w)
self.t2c = t2c
self.t2f = t2f
self.t2g = t2g
self.t2s = t2s
self.t2w = t2w

self.original_text = text
self.disable_ligatures = disable_ligatures
text_without_tabs = text
if text.find("\t") != -1:
    text_without_tabs = text.replace("\t", " " * self.tab_width)
self.text = text_without_tabs
if self.line_spacing == -1:
    self.line_spacing = (
        self._font_size + self._font_size * DEFAULT_LINE_SPACING_SCALE
    )
else:
    self.line_spacing = self._font_size + self._font_size * self.line_spacing

color = Color(color) if color else VMobject().color
file_name = self._text2svg(color)
PangoUtils.remove_last_M(file_name)
super().__init__(
    file_name,
    fill_opacity=fill_opacity,
    stroke_width=stroke_width,
    height=height,
    width=width,
    should_center=should_center,
    use_svg_cache=False,
    **kargs,
)
self.text = text
if self.disable_ligatures:
    self.subobjects = [*self._gen_chars()]
self.chars = self.get_group_class()(*self.subobjects)
self.text = text_without_tabs.replace(" ", "").replace("\n", "")
nppc = self.n_points_per_curve
for each in self:
    if len(each.points) == 0:
        continue
    points = each.points
    curve_start = points[0]
    assert len(curve_start) == self.dim, curve_start
    # Some of the glyphs in this text might not be closed,
    # so we close them by identifying when one curve ends
    # but it is not where the next curve starts.
    # It is more efficient to temporarily create a list
    # of points and add them one at a time, then turn them
    # into a numpy array at the end, rather than creating
    # new numpy arrays every time a point or fixing line
    # is added (which is O(n^2) for numpy arrays).
    closed_curve_points = []
    # OpenGL has points be part of quadratic Bezier curves;
    # Cairo uses cubic Bezier curves.
    if nppc == 3: # RendererType.OPENGL

        def add_line_to(end):
            nonlocal closed_curve_points
            start = closed_curve_points[-1]
            closed_curve_points += [
                start,
                (start + end) / 2,
                end,
            ]
    else: # RendererType.CAIRO

        def add_line_to(end):
            nonlocal closed_curve_points
            start = closed_curve_points[-1]
            closed_curve_points += [
                start,
                (start + start + end) / 3,
                (start + end + end) / 3,
                end,
            ]
    for index, point in enumerate(points):
        closed_curve_points.append(point)
        if (
            index != len(points) - 1
            and (index + 1) % nppc == 0
            and any(point != points[index + 1])
        ):
            # Add straight line from last point on this curve to the
            # start point on the next curve. We represent the line
            # as a cubic bezier curve where the two control points
            # are half-way between the start and stop point.
            add_line_to(curve_start)
            curve_start = points[index + 1]
    # Make sure last curve is closed
    add_line_to(curve_start)
    each.points = np.array(closed_curve_points, ndmin=2)
# anti-aliasing
if height is None and width is None:
    self.scale(TEXT_MOB_SCALE_FACTOR)
self.initial_height = self.height

```

```

def __repr__(self):
    return f"Text({repr(self.original_text)})"

@property
def font_size(self):
    return (
        self.height
        / self.initial_height
        / TEXT_MOB_SCALE_FACTOR
        * 2.4
        * self._font_size
        / DEFAULT_FONT_SIZE
    )

@font_size.setter
def font_size(self, font_val):
    # TODO: use pango's font size scaling.
    if font_val <= 0:
        raise ValueError("font_size must be greater than 0.")
    else:
        self.scale(font_val / self.font_size)

def _gen_chars(self):
    chars = self.get_group_class()()
    subobjects_char_index = 0
    for char_index in range(len(self.text)):
        if self.text[char_index].isspace():
            space = Dot(radius=0, fill_opacity=0, stroke_opacity=0)
            if char_index == 0:
                space.move_to(self.subobjects[subobjects_char_index].get_center())
            else:
                space.move_to(
                    self.subobjects[subobjects_char_index - 1].get_center(),
                )
            chars.add(space)
        else:
            chars.add(self.subobjects[subobjects_char_index])
            subobjects_char_index += 1
    return chars

def find_indexes(self, word: str, text: str):
    """Finds the indexes of ``text`` in ``word``."""
    temp = re.match(r"\[(0-9-){0,}\]:([0-9-]{0,})\]", word)
    if temp:
        start = int(temp.group(1)) if temp.group(1) != "" else 0
        end = int(temp.group(2)) if temp.group(2) != "" else len(text)
        start = len(text) + start if start < 0 else start
        end = len(text) + end if end < 0 else end
        return [(start, end)]
    indexes = []
    index = text.find(word)
    while index != -1:
        indexes.append((index, index + len(word)))
        index = text.find(word, index + len(word))
    return indexes

@deprecated(
    since="v0.14.0",
    until="v0.15.0",
    message="This was internal function, you shouldn't be using it anyway.",
)
def _set_color_by_t2c(self, t2c=None):
    """Sets color for specified strings."""
    t2c = t2c if t2c else self.t2c
    for word, color in list(t2c.items()):
        for start, end in self._find_indexes(word, self.text):
            self.chars[start:end].set_color(color)

@deprecated(
    since="v0.14.0",
    until="v0.15.0",
    message="This was internal function, you shouldn't be using it anyway.",
)
def _set_color_by_t2g(self, t2g=None):
    """Sets gradient colors for specified strings.
    Behaves similarly to ``set_color_by_t2c``."""
    t2g = t2g if t2g else self.t2g
    for word, gradient in list(t2g.items()):
        for start, end in self._find_indexes(word, self.text):
            self.chars[start:end].set_color_by_gradient(*gradient)

def text2hash(self, color: Color):
    """Generates `sha256` hash for file name."""
    settings = (
        "PANGO" + self.font + self.slant + self.weight + color.hex_l
    ) # to differentiate Text and CairoText
    settings += str(self.t2f) + str(self.t2s) + str(self.t2w) + str(self.t2c)
    settings += str(self.line_spacing) + str(self._font_size)
    settings += str(self.disable_ligatures)
    id_str = self.text + settings
    hasher = hashlib.sha256()
    hasher.update(id_str.encode())
    return hasher.hexdigest()[:16]

def _merge_settings(
    self,
    left_setting: TextSetting,
    right_setting: TextSetting,
    default_args: dict[str, Iterable[str]],
) -> TextSetting:
    contained = right_setting.end < left_setting.end
    new_setting = copy.copy(left_setting) if contained else copy.copy(right_setting)

    new_setting.start = right_setting.end if contained else left_setting.end
    left_setting.end = right_setting.start
    if not contained:
        right_setting.end = new_setting.start

    for arg in default_args:
        left = getattr(left_setting, arg)
        right = getattr(right_setting, arg)
        default = default_args[arg]
        if left != default and getattribute(right_setting, arg) != default:
            raise ValueError(

```

```

        f"Ambiguous style for text '{self.text[right_setting.start:right_setting.end]}':"
        + f"'{arg}' cannot be both '{left}' and '{right}'."
    )
    setattr(right_setting, arg, left if left != default else right)
return new_setting

def _get_settings_from_t2xs(
    self,
    t2xs: Sequence[tuple[dict[str, str], str]],
    default_args: dict[str, Iterable[str]],
) -> Sequence[TextSetting]:
    settings = []
    t2xwords = set(chain(*([*t2x.keys()] for t2x, _ in t2xs)))
    for word in t2xwords:
        setting_args = {
            arg: t2x[word] if word in t2x else default_args[arg]
            for t2x, arg in t2xs
        }

        for start, end in self._find_indexes(word, self.text):
            settings.append(TextSetting(start, end, **setting_args))
    return settings

def _get_settings_from_gradient(
    self,
    default_args: dict[str, Iterable[str]],
) -> Sequence[TextSetting]:
    settings = []
    args = copy.copy(default_args)
    if self.gradient:
        colors = color_gradient(self.gradient, len(self.text))
        for i in range(len(self.text)):
            args["color"] = colors[i].hex
            settings.append(TextSetting(i, i + 1, **args))

    for word, gradient in self.t2g.items():
        if isinstance(gradient, str) or len(gradient) == 1:
            color = gradient if isinstance(gradient, str) else gradient[0]
            gradient = [Color(color)]
        colors = (
            color_gradient(gradient, len(word))
            if len(gradient) != 1
            else len(word) * gradient
        )
        for start, end in self._find_indexes(word, self.text):
            for i in range(start, end):
                args["color"] = colors[i - start].hex
                settings.append(TextSetting(i, i + 1, **args))
    return settings

def _text2settings(self, color: Color):
    """Converts the texts and styles to a setting for parsing."""
    t2xs = [
        (self.t2f, "font"),
        (self.t2s, "slant"),
        (self.t2w, "weight"),
        (self.t2c, "color"),
    ]
    # setting_args requires values to be strings
    default_args = {
        arg: getattr(self, arg) if arg != "color" else str(color) for _, arg in t2xs
    }

    settings = self._get_settings_from_t2xs(t2xs, default_args)
    settings.extend(self._get_settings_from_gradient(default_args))

    # Handle overlaps

    settings.sort(key=lambda setting: setting.start)
    for index, setting in enumerate(settings):
        if index + 1 == len(settings):
            break

        next_setting = settings[index + 1]
        if setting.end > next_setting.start:
            new_setting = self._merge_settings(setting, next_setting, default_args)
            new_index = index + 1
            while (
                new_index < len(settings)
                and settings[new_index].start < new_setting.start
            ):
                new_index += 1
            settings.insert(new_index, new_setting)

    # Set all text settings (default font, slant, weight)
    temp_settings = settings.copy()
    start = 0
    for setting in settings:
        if setting.start != start:
            temp_settings.append(TextSetting(start, setting.start, **default_args))
            start = setting.end
    if start != len(self.text):
        temp_settings.append(TextSetting(start, len(self.text), **default_args))
    settings = sorted(temp_settings, key=lambda setting: setting.start)

    line_num = 0
    if re.search(r"\n", self.text):
        for start, end in self._find_indexes("\n", self.text):
            for setting in settings:
                if setting.line_num == -1:
                    setting.line_num = line_num
                if start < setting.end:
                    line_num += 1
                    new_setting = copy.copy(setting)
                    setting.end = end
                    new_setting.start = end
                    new_setting.line_num = line_num
                    settings.append(new_setting)
                    settings.sort(key=lambda setting: setting.start)
                    break
    for setting in settings:
        if setting.line_num == -1:
            setting.line_num = line_num
    return settings

```

```

def text2svg(self, color: Color):
    """Convert the text to SVG using Pango."""
    size = self._font_size
    line_spacing = self.line_spacing
    size /= TEXT2SVG_ADJUSTMENT_FACTOR
    line_spacing /= TEXT2SVG_ADJUSTMENT_FACTOR

    dir_name = config.get_dir("text_dir")
    if not dir_name.is_dir():
        dir_name.mkdir(parents=True)
    hash_name = self._text2hash(color)
    file_name = dir_name / (hash_name + ".svg")

    if file_name.exists():
        svg_file = str(file_name.resolve())
    else:
        settings = self._text2settings(color)
        width = config["pixel_width"]
        height = config["pixel_height"]

        svg_file = manimpango.text2svg(
            settings,
            size,
            line_spacing,
            self.disable_ligatures,
            str(file_name.resolve()),
            START_X,
            START_Y,
            width,
            height,
            self.text,
        )

    return svg_file

def init_colors(self, propagate_colors=True):
    if config.renderer == RendererType.OPENGL:
        super().init_colors()
    elif config.renderer == RendererType.CAIRO:
        super().init_colors(propagate_colors=propagate_colors)

class MarkupText(SVGObject):
    """Display (non-LaTeX) text rendered using `Pango <https://pango.gnome.org/>`_.

    Text objects behave like a :class:`.VGroup`-like iterable of all characters
    in the given text. In particular, slicing is possible.

    **What is PangоМаркап?**

    PangоМаркап is a small markup language like html and it helps you avoid using
    "range of characters" while coloring or styling a piece a Text. You can use
    this language with :class:`~.MarkupText`.

    A simple example of a marked-up string might be::

        <span foreground="blue" size="x-large">Blue text</span> is <i>cool</i>!

    and it can be used with :class:`~.MarkupText` as

    .. manim:: MarkupExample
        :save_last_frame:

        class MarkupExample(Scene):
            def construct(self):
                text = MarkupText('<span foreground="blue" size="x-large">Blue text</span> is <i>cool</i>!')
                self.add(text)

    A more elaborate example would be:

    .. manim:: MarkupElaborateExample
        :save_last_frame:

        class MarkupElaborateExample(Scene):
            def construct(self):
                text = MarkupText(
                    '<span foreground="purple">Ø§</span><span foreground="red">Ùž</span>'
                    'Ù„<span foreground="blue">Ù‘</span>Ø¹<span foreground="red">Ùž</span>Ø±'
                    '<span foreground="red">Ùž</span>Ø“<span foreground="red">Ù‘</span>Ùš'
                    '<span foreground="green">Ù‘</span><span foreground="red">Ùž</span>Ø®'
                    '<span foreground="blue">Ù‘</span>'
                )
                self.add(text)

    PangоМаркап can also contain XML features such as numeric character
    entities such as ``&#169;`` for ® can be used too.

    The most general markup tag is ``<span>``, then there are some
    convenience tags.

    Here is a list of supported tags:

    - ``<b>bold</b>``, ``<i>italic</i>`` and ``<b><i>bold+italic</i></b>``
    - ``<u>underline</u>`` and ``<s>strike through</s>``
    - ``<tt>typewriter font</tt>``
    - ``<big>bigger font</big>`` and ``<small>smaller font</small>``
    - ``<sup>superscript</sup>`` and ``<sub>subscript</sub>``
    - ``<span underline="double" underline_color="green">double underline</span>``
    - ``<span underline="error">error underline</span>``
    - ``<span overline="single" overline_color="green">overline</span>``
    - ``<span strikethrough="true" strikethrough_color="red">strikethrough</span>``
    - ``<span font_family="sans">temporary change of font</span>``
    - ``<span foreground="red">temporary change of color</span>``
    - ``<span fgcolor="red">temporary change of color</span>``
    - ``<gradient from="YELLOW" to="RED">temporary gradient</gradient>``

    For ``<span>`` markup, colors can be specified either as
    hex triples like ``#aabbcc`` or as named CSS colors like
    ``AliceBlue``.

    The ``<gradient>`` tag is handled by Manim rather than
    Pango, and supports hex triplets or Manim constants like
    ``RED`` or ``RED_A``.

    If you want to use Manim constants like ``RED_A`` together
```

with ````, you will need to use Python's f-String syntax as follows::

```
MarkupText(f'<span foreground="(RED_A)">here you go</span>')
```

If your text contains ligatures, the `:class:`MarkupText`` class may incorrectly determine the first and last letter when creating the gradient. This is due to the fact that ``fl`` are two separate characters, but might be set as one single glyph - a ligature. If your language does not depend on ligatures, consider setting ```disable_ligatures``` to ``True``. If you must use ligatures, the ```gradient``` tag supports an optional attribute ```offset``` which can be used to compensate for that error.

For example:

```
- ``<gradient from="RED" to="YELLOW" offset="1">example</gradient>`` to *start* the gradient one letter earlier
- ``<gradient from="RED" to="YELLOW" offset=",1">example</gradient>`` to *end* the gradient one letter earlier
- ``<gradient from="RED" to="YELLOW" offset="2,1">example</gradient>`` to *start* the gradient two letters earlier and *end* it one letter earlier
```

Specifying a second offset may be necessary if the text to be colored does itself contain ligatures. The same can happen when using HTML entities for special chars.

When using ```underline```, ```overline``` or ```strikethrough``` together with ```<gradient>``` tags, you will also need to use the `offset`, because underlines are additional paths in the final `:class:`SVGObject``. Check out the following example.

Escaping of special characters: ``>`` `**should**` be written as ``>`` whereas ``<`` and ``&`` `*must*` be written as ``<`` and ``&``.

You can find more information about Pango markup formatting at the corresponding documentation page:
[Pango Markup <https://docs.gtk.org/Pango/pango_markup.html>](https://docs.gtk.org/Pango/pango_markup.html). Please be aware that not all features are supported by this class and that the ```<gradient>``` tag mentioned above is not supported by Pango.

Parameters

`text`
The text that needs to be created as mobject.
`fill_opacity`
The fill opacity, with 1 meaning opaque and 0 meaning transparent.
`stroke_width`
Stroke width.
`font_size`
Font size.
`line_spacing`
Line spacing.
`font`
Global font setting for the entire text. Local overrides are possible.
`slant`
Global slant setting, e.g. `NORMAL` or `ITALIC`. Local overrides are possible.
`weight`
Global weight setting, e.g. `NORMAL` or `BOLD`. Local overrides are possible.
`gradient`
Global gradient setting. Local overrides are possible.
`warn_missing_font`
If True (default), Manim will issue a warning if the font does not exist in the (case-sensitive) list of fonts returned from `manimpango.list_fonts()`.

Returns

`:class:`MarkupText``
The text displayed in form of a `:class:`.VGroup``-like mobject.

Examples

```
.. manim:: BasicMarkupExample
    :save_last_frame:

    class BasicMarkupExample(Scene):
        def construct(self):
            text1 = MarkupText("<b>foo</b> <i>bar</i> <b><i>foobar</i></b>")
            text2 = MarkupText("<s>foo</s> <u>bar</u> <big>big</big> <small>small</small>")
            text3 = MarkupText("<sub>2</sub> and <sub>3</sub>+<sup>0</sup>-<sup>1</sup>")
            text4 = MarkupText("type <tt>help</tt> for help")
            text5 = MarkupText(
                '<span underline="double">foo</span> <span underline="error">bar</span>'
            )
            group = VGroup(text1, text2, text3, text4, text5).arrange(DOWN)
            self.add(group)

.. manim:: ColorExample
    :save_last_frame:

    class ColorExample(Scene):
        def construct(self):
            text1 = MarkupText(
                'all in red <span fgcolor="#YELLOW">except this</span>', color=RED
            )
            text2 = MarkupText("nice gradient", gradient=(BLUE, GREEN))
            text3 = MarkupText(
                'nice <gradient from="RED" to="YELLOW">intermediate</gradient> gradient',
                gradient=(BLUE, GREEN),
            )
            text4 = MarkupText(
                'fl ligature <gradient from="RED" to="YELLOW">causing trouble</gradient> here'
            )
            text5 = MarkupText(
                'fl ligature <gradient from="RED" to="YELLOW" offset="1">defeated</gradient> with offset'
            )
            text6 = MarkupText(
                'fl ligature <gradient from="RED" to="YELLOW" offset="1">floating</gradient> inside'
            )
            text7 = MarkupText(
                'fl ligature <gradient from="RED" to="YELLOW" offset="1,1">floating</gradient> inside'
            )
            group = VGroup(text1, text2, text3, text4, text5, text6, text7).arrange(DOWN)
            self.add(group)
```

```

.. manim:: UnderlineExample
:save_last_frame:

class UnderlineExample(Scene):
    def construct(self):
        text1 = MarkupText(
            '<span underline="double" underline_color="green">bla</span>'
        )
        text2 = MarkupText(
            '<span underline="single" underline_color="green">xxxx</span><gradient from="#ffff00" to="RED">aabb</gradient>y'
        )
        text3 = MarkupText(
            '<span underline="single" underline_color="green">xxxx</span><gradient from="#ffff00" to="RED" offset="-1">aabb</gradient>y'
        )
        text4 = MarkupText(
            '<span underline="double" underline_color="green">xxxx</span><gradient from="#ffff00" to="RED">aabb</gradient>y'
        )
        text5 = MarkupText(
            '<span underline="double" underline_color="green">xxxx</span><gradient from="#ffff00" to="RED" offset="-2">aabb</gradient>y'
        )
        group = VGroup(text1, text2, text3, text4, text5).arrange(DOWN)
        self.add(group)

.. manim:: FontExample
:save_last_frame:

class FontExample(Scene):
    def construct(self):
        text1 = MarkupText(
            'all in sans <span font_family="serif">except this</span>, font="sans"'
        )
        text2 = MarkupText(
            '<span font_family="serif">mixing</span> <span font_family="sans">fonts</span> <span font_family="monospace">is ugly</span>'
        )
        text3 = MarkupText("special char > or &gt;")
        text4 = MarkupText("special char &lt; and &amp;")
        group = VGroup(text1, text2, text3, text4).arrange(DOWN)
        self.add(group)

.. manim:: NewlineExample
:save_last_frame:

class NewlineExample(Scene):
    def construct(self):
        text = MarkupText('foooo<span foreground="red">oo\nbaa</span>aar')
        self.add(text)

.. manim:: NoLigaturesExample
:save_last_frame:

class NoLigaturesExample(Scene):
    def construct(self):
        text1 = MarkupText('fl<gradient from="RED" to="GREEN">oat</gradient>ing')
        text2 = MarkupText('fl<gradient from="RED" to="GREEN">oat</gradient>ing', disable_ligatures=True)
        group = VGroup(text1, text2).arrange(DOWN)
        self.add(group)

```

As `:class:`MarkupText`` uses Pango to render text, rendering non-English characters is easily possible:

```

.. manim:: MultiLanguage
:save_last_frame:

class MultiLanguage(Scene):
    def construct(self):
        morning = MarkupText("à@ù@è@•à~à@•à@à~", font="sans-serif")
        japanese = MarkupText(
            '<span fgcolor="blue">æ~ÿœ~</span>ä,ä,^ätä"ä'
        ) # works as in ``Text``.
        mess = MarkupText("Multi-Language", weight=BOLD)
        russ = MarkupText("Д'Н'ЕД"ї?Н,Д'Н'ї,Д'Н'ї, а"ї@ а@ ", font="sans-serif")
        hin = MarkupText("à~à@à, à~à@à~!", font="sans-serif")
        chinese = MarkupText("è†,ç‡æ€é»›æ~"äcå,¶é~'ä@å", font="sans-serif")
        group = VGroup(morning, japanese, mess, russ, hin, chinese).arrange(DOWN)
        self.add(group)

```

You can justify the text by passing `:attr:`justify`` parameter.

```

.. manim:: JustifyText

class JustifyText(Scene):
    def construct(self):
        ipsum_text = (
            "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
            "Praesent feugiat metus sit amet iaculis pulvinar. Nulla posuere "
            "quam a ex aliquam, eleifend consectetur tellus viverra. Aliquam "
            "fermentum interdum justo, nec rutrum elit pretium ac. Nam quis "
            "leo pulvinar, dignissim est at, venenatis nisi."
        )
        justified_text = MarkupText(ipsum_text, justify=True).scale(0.4)
        not_justified_text = MarkupText(ipsum_text, justify=False).scale(0.4)
        just_title = Title("Justified")
        njust_title = Title("Not Justified")
        self.add(njust_title, not_justified_text)
        self.play(
            FadeOut(not_justified_text),
            FadeIn(justified_text),
            FadeOut(njust_title),
            FadeIn(just_title),
        )
        self.wait(1)

```

Tests

Check that the creation of `:class:`~.MarkupText`` works::

```
>>> MarkupText('The horse does not eat cucumber salad.')
MarkupText('The horse does not eat cucumber salad.')
```

"""

```
def __init__(
```

```

self,
text: str,
fill_opacity: float = 1,
stroke_width: float = 0,
color: Color | None = None,
font_size: float = DEFAULT_FONT_SIZE,
line_spacing: int = -1,
font: str = "",
slant: str = NORMAL,
weight: str = NORMAL,
justify: bool = False,
gradient: tuple = None,
tab_width: int = 4,
height: int = None,
width: int = None,
should_center: bool = True,
disable_ligatures: bool = False,
warn_missing_font: bool = True,
**kwargs,
) -> None:
    self.text = text
    self.line_spacing = line_spacing
    if font and warn_missing_font:
        fonts_list = manimpango.list_fonts()
        if font not in fonts_list:
            logger.warning(f"Font {font} not in {fonts_list}.")
    self.font = font
    self._font_size = float(font_size)
    self.slant = slant
    self.weight = weight
    self.gradient = gradient
    self.tab_width = tab_width
    self.justify = justify

    self.original_text = text
    self.disable_ligatures = disable_ligatures
    text_without_tabs = text
    if "\t" in text:
        text_without_tabs = text.replace("\t", " " * self.tab_width)

    colormap = self._extract_color_tags()
    if len(colormap) > 0:
        logger.warning(
            'Using <color> tags in MarkupText is deprecated. Please use <span foreground="..."> instead.',
        )
    gradientmap = self._extract_gradient_tags()
    validate_error = MarkupUtils.validate(self.text)
    if validate_error:
        raise ValueError(validate_error)

    if self.line_spacing == -1:
        self.line_spacing = (
            self._font_size + self._font_size * DEFAULT_LINE_SPACING_SCALE
        )
    else:
        self.line_spacing = self._font_size + self._font_size * self.line_spacing

    color = Color(color) if color else VMobject().color
    file_name = self._text2svg(color)

    PangoUtils.remove_last_M(file_name)
    super().__init__(
        file_name,
        fill_opacity=fill_opacity,
        stroke_width=stroke_width,
        height=height,
        width=width,
        should_center=should_center,
        **kwargs,
    )

    self.chars = self.get_group_class()(*self.subobjects)
    self.text = text_without_tabs.replace(" ", "").replace("\n", "")

nppc = self.n_points_per_curve
for each in self:
    if len(each.points) == 0:
        continue
    points = each.points
    curve_start = points[0]
    assert len(curve_start) == self.dim, curve_start
    # Some of the glyphs in this text might not be closed,
    # so we close them by identifying when one curve ends
    # but it is not where the next curve starts.
    # It is more efficient to temporarily create a list
    # of points and add them one at a time, then turn them
    # into a numpy array at the end, rather than creating
    # new numpy arrays every time a point or fixing line
    # is added (which is O(n^2) for numpy arrays).
    closed_curve_points = []
    # OpenGL has points be part of quadratic Bezier curves;
    # Cairo uses cubic Bezier curves.
    if nppc == 3: # RenderType.OPENGL

        def add_line_to(end):
            nonlocal closed_curve_points
            start = closed_curve_points[-1]
            closed_curve_points += [
                start,
                (start + end) / 2,
                end,
            ]
    else: # RenderType.CAIRO

        def add_line_to(end):
            nonlocal closed_curve_points
            start = closed_curve_points[-1]
            closed_curve_points += [
                start,
                (start + start + end) / 3,
                (start + end + end) / 3,
                end,
            ]

```

```

        for index, point in enumerate(points):
            closed_curve_points.append(point)
            if (
                index != len(points) - 1
                and (index + 1) % npcc == 0
                and any(point != points[index + 1])
            ):
                # Add straight line from last point on this curve to the
                # start point on the next curve.
                add_line_to(curve_start)
                curve_start = points[index + 1]
        # Make sure last curve is closed
        add_line_to(curve_start)
        each.points = np.array(closed_curve_points, ndmin=2)

    if self.gradient:
        self.set_color_by_gradient(*self.gradient)
    for col in colormap:
        self.chars[
            col["start"]
            - col["start_offset"] : col["end"]
            - col["start_offset"]
            - col["end_offset"]
        ].set_color(self._parse_color(col["color"]))
    for grad in gradientmap:
        self.chars[
            grad["start"]
            - grad["start_offset"] : grad["end"]
            - grad["start_offset"]
            - grad["end_offset"]
        ].set_color_by_gradient(
            *(self._parse_color(grad["from"]), self._parse_color(grad["to"]))
        )
    # anti-aliasing
    if height is None and width is None:
        self.scale(TEXT_MOB_SCALE_FACTOR)

    self.initial_height = self.height

@property
def font_size(self):
    return (
        self.height
        / self.initial_height
        / TEXT_MOB_SCALE_FACTOR
        * 2.4
        * self._font_size
        / DEFAULT_FONT_SIZE
    )

@font_size.setter
def font_size(self, font_val):
    # TODO: use pango's font size scaling.
    if font_val <= 0:
        raise ValueError("font_size must be greater than 0.")
    else:
        self.scale(font_val / self.font_size)

def _text2hash(self, color: Color):
    """Generates ``sha256`` hash for file name."""
    settings = (
        "MARKUPPANGO" + self.font + self.slant + self.weight + color.hex_1
    ) # to differentiate from classical Pango Text
    settings += str(self.line_spacing) + str(self._font_size)
    settings += str(self.disable_ligatures)
    settings += str(self.justify)
    id_str = self.text + settings
    hasher = hashlib.sha256()
    hasher.update(id_str.encode())
    hasher.update(id_str.encode())
    return hasher.hexdigest()[:16]

def _text2svg(self, color: Color | None):
    """Convert the text to SVG using Pango."""
    size = self._font_size
    line_spacing = self.line_spacing
    size /= TEXT2SVG_ADJUSTMENT_FACTOR
    line_spacing /= TEXT2SVG_ADJUSTMENT_FACTOR

    dir_name = config.get_dir("text_dir")
    if not dir_name.exists():
        dir_name.mkdir(parents=True)
    hash_name = self._text2hash(color)
    file_name = dir_name / (hash_name + ".svg")
    if file_name.exists():
        svg_file = str(file_name.resolve())
    else:
        final_text = (
            f"<span foreground=\"{color}\">{self.text}</span>"
            if color is not None
            else self.text
        )
        logger.debug(f"Setting Text {self.text}")
        svg_file = MarkupUtils.text2svg(
            final_text,
            self.font,
            self.slant,
            self.weight,
            size,
            line_spacing,
            self.disable_ligatures,
            str(file_name.resolve()),
            START_X,
            START_Y,
            600, # width
            400, # height
            justify=self.justify,
            pango_width=500,
        )
    return svg_file

def _count_real_chars(self, s):
    """Counts characters that will be displayed.

```

```

This is needed for partial coloring or gradients, because space
counts to the text's `len`, but has no corresponding character."""
count = 0
level = 0
# temporarily replace HTML entities by single char
s = re.sub("&[^;]+;", "x", s)
for c in s:
    if c == "<":
        level += 1
    if c == ">" and level > 0:
        level -= 1
    elif c != " " and c != "\t" and level == 0:
        count += 1
return count

def _extract_gradient_tags(self):
    """Used to determine which parts (if any) of the string should be formatted
    with a gradient.

    Removes the ``<gradient>`` tag, as it is not part of Pango's markup and would cause an error.
    """
    tags = re.finditer(
        r'<gradient>(?P<from>[^"]+) "(?P<to>[^"]+)" (?P<offset>[^"]+)?>(.+?)</gradient>',
        self.original_text,
        re.S,
    )
    gradientmap = []
    for tag in tags:
        start = self._count_real_chars(self.original_text[: tag.start(0)])
        end = start + self._count_real_chars(tag.group(5))
        offsets = tag.group(4).split(",") if tag.group(4) else [0]
        start_offset = int(offsets[0]) if offsets[0] else 0
        end_offset = int(offsets[1]) if len(offsets) == 2 and offsets[1] else 0
        gradientmap.append(
            {
                "start": start,
                "end": end,
                "from": tag.group(1),
                "to": tag.group(2),
                "start_offset": start_offset,
                "end_offset": end_offset,
            },
        )
    self.text = re.sub("<gradient[^>]+>(.+?)</gradient>", r"\1", self.text, 0, re.S)
    return gradientmap

def _parse_color(self, col):
    """Parse color given in ``<color>`` or ``<gradient>`` tags."""
    if re.match("#[0-9a-f]{6}", col):
        return col
    else:
        return Colors[col.lower()].value

def _extract_color_tags(self):
    """Used to determine which parts (if any) of the string should be formatted
    with a custom color.

    Removes the ``<color>`` tag, as it is not part of Pango's markup and would cause an error.

    Note: Using the ``<color>`` tags is deprecated. As soon as the legacy syntax is gone, this function
    will be removed.
    """
    tags = re.finditer(
        r'<color>(?P<col>[^"]+) "(?P<offset>[^"]+)?>(.+?)</color>',
        self.original_text,
        re.S,
    )
    colormap = []
    for tag in tags:
        start = self._count_real_chars(self.original_text[: tag.start(0)])
        end = start + self._count_real_chars(tag.group(4))
        offsets = tag.group(3).split(",") if tag.group(3) else [0]
        start_offset = int(offsets[0]) if offsets[0] else 0
        end_offset = int(offsets[1]) if len(offsets) == 2 and offsets[1] else 0
        colormap.append(
            {
                "start": start,
                "end": end,
                "color": tag.group(1),
                "start_offset": start_offset,
                "end_offset": end_offset,
            },
        )
    self.text = re.sub("<color[^>]+>(.+?)</color>", r"\1", self.text, 0, re.S)
    return colormap

def __repr__(self):
    return f"MarkupText({repr(self.original_text)})"


```

```

@contextmanager
def register_font(font_file: str | Path):
    """Temporarily add a font file to Pango's search path.

    This searches for the font_file at various places. The order it searches is described below.

```

1. Absolute path.
2. In ``assets/fonts`` folder.
3. ``font/`` folder.
4. In the same directory.

Parameters

font_file

The font file to add.

Examples

Use ``with register_font(...)`` to add a font file to search path.

```

.. code-block:: python

    with register_font("path/to/font_file.ttf"):
        a = Text("Hello", font="Custom Font Name")

Raises
-----
FileNotFoundException:
    If the font doesn't exists.

AttributeError:
    If this method is used on macOS.

.. important ::

    This method is available for macOS for ``ManimPango>=v0.2.3``. Using this
    method with previous releases will raise an :class:`AttributeError` on macOS.

"""

input_folder = Path(config.input_file).parent.resolve()
possible_paths = [
    Path(font_file),
    input_folder / "assets/fonts" / font_file,
    input_folder / "fonts" / font_file,
    input_folder / font_file,
]
for path in possible_paths:
    path = path.resolve()
    if path.exists():
        file_path = path
        logger.debug("Found file at %s", file_path.absolute())
        break
    else:
        error = f"Can't find {font_file}." f"Tried these : {possible_paths}"
        raise FileNotFoundError(error)

try:
    assert manimpango.register_font(str(file_path))
    yield
finally:
    manimpango.unregister_font(str(file_path))

```

opengl_three_dimensions.py

```
from __future__ import annotations

import numpy as np

from manim.mobject.opengl.opengl_surface import OpenGLSurface
from manim.mobject.opengl_vectorized_mobject import OpenGLGroup, OpenGLMobject

class OpenGLSurfaceMesh(OpenGLGroup):
    def __init__(
        self,
        uv_surface,
        resolution=None,
        stroke_width=1,
        normal_nudge=1e-2,
        depth_test=True,
        flat_stroke=False,
        **kwargs,
    ):
        if not isinstance(uv_surface, OpenGLSurface):
            raise Exception("uv_surface must be of type OpenGLSurface")
        self.uv_surface = uv_surface
        self.resolution = resolution if resolution is not None else (21, 21)
        self.normal_nudge = normal_nudge
        super().__init__(
            stroke_width=stroke_width,
            depth_test=depth_test,
            flat_stroke=flat_stroke,
            **kwargs,
        )

    def init_points(self):
        uv_surface = self.uv_surface

        full_nu, full_nv = uv_surface.resolution
        part_nu, part_nv = self.resolution
        u_indices = np.linspace(0, full_nu, part_nu).astype(int)
        v_indices = np.linspace(0, full_nv, part_nv).astype(int)

        points, du_points, dv_points = uv_surface.get_surface_points_and_nudged_points()
        normals = uv_surface.get_unit_normals()
        nudged_points = points + self.normal_nudge * normals

        for ui in u_indices:
            path = OpenGLMobject()
            full_ui = full_nv * ui
            path.set_points_smoothly(nudged_points[full_ui : full_ui + full_nv])
            self.add(path)
        for vi in v_indices:
            path = OpenGLMobject()
            path.set_points_smoothly(nudged_points[vi::full_nv])
            self.add(path)
```

numbers.py_1

```
"""Mobjects representing numbers."""

from __future__ import annotations

__all__ = ["DecimalNumber", "Integer", "Variable"]

from typing import Sequence

import numpy as np

from manim import config
from manim.constants import *
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.text.tex_mobject import MathTex, SingleStringMathTex, Tex
from manim.mobject.text.text_mobject import Text
from manim.mobject.types.vectorized_mobject import VMobject
from manim.mobject.value_tracker import ValueTracker

string_to_mob_map = {}

class DecimalNumber(VMobject, metaclass=ConvertToOpenGL):
    """An mobject representing a decimal number.

    Examples
    -------

    .. manim:: MovingSquareWithUpdaters

        class MovingSquareWithUpdaters(Scene):
            def construct(self):
                decimal = DecimalNumber(
                    0,
                    show_ellipsis=True,
                    num_decimal_places=3,
                    include_sign=True,
                )
                square = Square().to_edge(UP)

                decimal.add_updater(lambda d: d.next_to(square, RIGHT))
                decimal.add_updater(lambda d: d.set_value(square.get_center()[1]))
                self.add(square, decimal)
                self.play(
                    square.animate.to_edge(DOWN),
                    rate_func=there_and_back,
                    run_time=5,
                )
                self.wait()

    """

    def __init__(
        self,
        number: float = 0,
        num_decimal_places: int = 2,
        mob_class: VMobject = MathTex,
        include_sign: bool = False,
        group_with_commas: bool = True,
        digit_buff_per_font_unit: float = 0.001,
        show_ellipsis: bool = False,
        unit: str | None = None, # Aligned to bottom unless it starts with "^"
        include_background_rectangle: bool = False,
        edge_to_fix: Sequence[float] = LEFT,
        font_size: float = DEFAULT_FONT_SIZE,
        stroke_width: float = 0,
        fill_opacity: float = 1.0,
        **kwargs,
    ):
        super().__init__(**kwargs, stroke_width=stroke_width)
        self.number = number
        self.num_decimal_places = num_decimal_places
        self.include_sign = include_sign
        self.mob_class = mob_class
        self.group_with_commas = group_with_commas
        self.digit_buff_per_font_unit = digit_buff_per_font_unit
        self.show_ellipsis = show_ellipsis
        self.unit = unit
        self.include_background_rectangle = include_background_rectangle
        self.edge_to_fix = edge_to_fix
        self._font_size = font_size
        self.fill_opacity = fill_opacity
```

```

self.initial_config = kwargs.copy()
self.initial_config.update(
{
    "num_decimal_places": num_decimal_places,
    "include_sign": include_sign,
    "group_with_commas": group_with_commas,
    "digit_buff_per_font_unit": digit_buff_per_font_unit,
    "show_ellipsis": show_ellipsis,
    "unit": unit,
    "include_background_rectangle": include_background_rectangle,
    "edge_to_fix": edge_to_fix,
    "font_size": font_size,
    "stroke_width": stroke_width,
    "fill_opacity": fill_opacity,
},
)

self._set_submobjects_from_number(number)
self.init_colors()

@property
def font_size(self):
    """The font size of the tex mobject."""
    return self.height / self.initial_height * self._font_size

@font_size.setter
def font_size(self, font_val):
    if font_val <= 0:
        raise ValueError("font_size must be greater than 0.")
    elif self.height > 0:
        # sometimes manim generates a SingleStringMathTex mobject with 0 height.
        # can't be scaled regardless and will error without the elif.

        # scale to a factor of the initial height so that setting
        # font_size does not depend on current size.
        self.scale(font_val / self.font_size)

def _set_submobjects_from_number(self, number):
    self.number = number
    self.submobjects = []

    num_string = self._get_num_string(number)
    self.add(*map(self._string_to_mob, num_string))

    # Add non-numerical bits
    if self.show_ellipsis:
        self.add(
            self._string_to_mob("\dots", SingleStringMathTex, color=self.color),
        )

    if self.unit is not None:
        self.unit_sign = self._string_to_mob(self.unit, SingleStringMathTex)
        self.add(self.unit_sign)

    self.arrange(
        buff=self.digit_buff_per_font_unit * self._font_size,
        aligned_edge=DOWN,
    )

    # Handle alignment of parts that should be aligned
    # to the bottom
    for i, c in enumerate(num_string):
        if c == "-" and len(num_string) > i + 1:
            self[i].align_to(self[i + 1], UP)
            self[i].shift(self[i + 1].height * DOWN / 2)
        elif c == ",":
            self[i].shift(self[i].height * DOWN / 2)
    if self.unit and self.unit.startswith("^"):
        self.unit_sign.align_to(self, UP)

    # track the initial height to enable scaling via font_size
    self.initial_height = self.height

    if self.include_background_rectangle:
        self.add_background_rectangle()

def _get_num_string(self, number):
    if isinstance(number, complex):
        formatter = self._get_complex_formatter()
    else:
        formatter = self._get_formatter()
    num_string = formatter.format(number)

    rounded_num = np.round(number, self.num_decimal_places)
    if num_string.startswith("-") and rounded_num == 0:
        if self.include_sign:
            num_string = "+" + num_string[1:]
        else:

```

```

        num_string = num_string[1:]

    return num_string

def _string_to_mob(self, string: str, mob_class: VMobject | None = None, **kwargs):
    if mob_class is None:
        mob_class = self.mob_class

    if string not in string_to_mob_map:
        string_to_mob_map[string] = mob_class(string, **kwargs)
    mob = string_to_mob_map[string].copy()
    mob.font_size = self._font_size
    return mob

def _get_formatter(self, **kwargs):
    """
    Configuration is based first off instance attributes,
    but overwritten by any keyword argument. Relevant
    key words:
    - include_sign
    - group_with_commas
    - num_decimal_places
    - field_name (e.g. 0 or 0.real)
    """
    config = {
        attr: getattr(self, attr)
        for attr in [
            "include_sign",
            "group_with_commas",
            "num_decimal_places",
        ]
    }
    config.update(kwargs)
    return "".join(
        [
            "{",
            config.get("field_name", ""),
            ":" ,
            "+" if config["include_sign"] else "",
            "," if config["group_with_commas"] else "",
            ".",
            str(config["num_decimal_places"]),
            "f",
            "}",
        ],
    )

def _get_complex_formatter(self):
    return "".join(
        [
            self._get_formatter(field_name="0.real"),
            self._get_formatter(field_name="0.imag", include_sign=True),
            "i",
        ],
    )

def set_value(self, number: float):
    """
    Set the value of the :class:`~.DecimalNumber` to a new number.

    Parameters
    -----
    number
        The value that will overwrite the current number of the :class:`~.DecimalNumber`.

    """
    # creates a new number mob via `set_submobjects_from_number`
    # then matches the properties (color, font_size, etc...)
    # of the previous mob object to the new one

    # old_family needed with cairo
    old_family = self.get_family()

    old_font_size = self.font_size
    move_to_point = self.get_edge_center(self.edge_to_fix)
    old_submobjects = self.submobjects

    self._set_submobjects_from_number(number)
    self.font_size = old_font_size
    self.move_to(move_to_point, self.edge_to_fix)
    for sm1, sm2 in zip(self.submobjects, old_submobjects):
        sm1.match_style(sm2)

    if config.renderer == RendererType.CAIRO:
        for mob in old_family:
            # Dumb hack...due to how scene handles families
            # of animated mobjects
            # for compatibility with updaters to not leave first number in place while updating,
            # not needed with opengl renderer

```

```

        mob.points[:] = 0

    self.init_colors()
    return self

def get_value(self):
    return self.number

def increment_value(self, delta_t=1):
    self.set_value(self.get_value() + delta_t)

class Integer(DecimalNumber):
    """A class for displaying Integers.

Examples
-----
.. manim:: IntegerExample
    :save_last_frame:

    class IntegerExample(Scene):
        def construct(self):
            self.add(Integer(number=2.5).set_color(ORANGE).scale(2.5).set_x(-0.5).set_y(0.8))
            self.add(Integer(number=3.14159, show_ellipsis=True).set_x(3).set_y(3.3).scale(3.14159))
            self.add(Integer(number=42).set_x(2.5).set_y(-2.3).set_color_by_gradient(BLUE, TEAL).scale(1.7))
            self.add(Integer(number=6.28).set_x(-1.5).set_y(-2).set_color(YELLOW).scale(1.4))
    """

def __init__(self, number=0, num_decimal_places=0, **kwargs):
    super().__init__(number=number, num_decimal_places=num_decimal_places, **kwargs)

def get_value(self):
    return int(np.round(super().get_value()))

class Variable(VMobject, metaclass=ConvertToOpenGL):
    """A class for displaying text that shows "label = value" with
    the value continuously updated from a :class:`ValueTracker`.

Parameters
-----
var
    The initial value you need to keep track of and display.
label
    The label for your variable. Raw strings are converted to :class:`MathTex` objects.
var_type
    The class used for displaying the number. Defaults to :class:`DecimalNumber`.
num_decimal_places
    The number of decimal places to display in your variable. Defaults to 2.
    If `var_type` is an :class:`Integer`, this parameter is ignored.
kwargs
    Other arguments to be passed to `~.Mobject`.

Attributes
-----
label : Union[:class:`str`, :class:`Tex`, :class:`MathTex`, :class:`Text`, :class:`SingleStringMathTex`]
    The label for your variable, for example ``x = ...``.
tracker : :class:`ValueTracker`
    Useful in updating the value of your variable on-screen.
value : Union[:class:`DecimalNumber`, :class:`Integer`]
    The tex for the value of your variable.

Examples
-----
Normal usage::

    # DecimalNumber type
    var = 0.5
    on_screen_var = Variable(var, Text("var"), num_decimal_places=3)
    # Integer type
    int_var = 0
    on_screen_int_var = Variable(int_var, Text("int_var"), var_type=Integer)
    # Using math mode for the label
    on_screen_int_var = Variable(int_var, "{a}_{i}", var_type=Integer)

.. manim:: VariablesWithValueTracker

    class VariablesWithValueTracker(Scene):
        def construct(self):
            var = 0.5
            on_screen_var = Variable(var, Text("var"), num_decimal_places=3)

            # You can also change the colours for the label and value
            on_screen_var.label.set_color(RED)
            on_screen_var.value.set_color(GREEN)

            self.play(Write(on_screen_var))

```

```

# The above line will just display the variable with
# its initial value on the screen. If you also wish to
# update it, you can do so by accessing the `tracker` attribute
self.wait()
var_tracker = on_screen_var.tracker
var = 10.5
self.play(var_tracker.animate.set_value(var))
self.wait()

int_var = 0
on_screen_int_var = Variable(
    int_var, Text("int_var"), var_type=Integer
).next_to(on_screen_var, DOWN)
on_screen_int_var.label.set_color(RED)
on_screen_int_var.value.set_color(GREEN)

self.play(Write(on_screen_int_var))
self.wait()
var_tracker = on_screen_int_var.tracker
var = 10.5
self.play(var_tracker.animate.set_value(var))
self.wait()

# If you wish to have a somewhat more complicated label for your
# variable with subscripts, superscripts, etc. the default class
# for the label is MathTex
subscript_label_var = 10
on_screen_subscript_var = Variable(subscript_label_var, "(a)_{i}").next_to(
    on_screen_int_var, DOWN
)
self.play(Write(on_screen_subscript_var))
self.wait()

.. manim:: VariableExample

class VariableExample(Scene):
    def construct(self):
        start = 2.0

        x_var = Variable(start, 'x', num_decimal_places=3)
        sqr_var = Variable(start**2, 'x^2', num_decimal_places=3)
        Group(x_var, sqr_var).arrange(DOWN)

        sqr_var.add_updater(lambda v: v.tracker.set_value(x_var.tracker.get_value()**2))

        self.add(x_var, sqr_var)
        self.play(x_var.tracker.animate.set_value(5), run_time=2, rate_func=linear)
        self.wait(0.1)

"""

def __init__(
    self,
    var: float,
    label: str | Tex | MathTex | Text | SingleStringMathTex,
    var_type: DecimalNumber | Integer = DecimalNumber,
    num_decimal_places: int = 2,
    **kwargs,
):
    self.label = MathTex(label) if isinstance(label, str) else label
    equals = MathTex("=").next_to(self.label, RIGHT)
    self.label.add>equals

    self.tracker = ValueTracker(var)

    if var_type == DecimalNumber:
        self.value = DecimalNumber(
            self.tracker.get_value(),
            num_decimal_places=num_decimal_places,
        )
    elif var_type == Integer:
        self.value = Integer(self.tracker.get_value())

    self.value.add_updater(lambda v: v.set_value(self.tracker.get_value())).next_to(
        self.label,
        RIGHT,
    )

super().__init__(**kwargs)
self.add(self.label, self.value)

```

paths.py

```
"""Functions determining transformation paths between sets of points."""
from __future__ import annotations

__all__ = [
    "straight_path",
    "path_along_arc",
    "clockwise_path",
    "counterclockwise_path",
]
from typing import Callable
import numpy as np

from ..constants import OUT
from ..utils.bezier import interpolate
from ..utils.deprecation import deprecated_params
from ..utils.space_ops import rotation_matrix

STRAIGHT_PATH_THRESHOLD = 0.01

PATH_FUNC_TYPE = Callable[[np.ndarray, np.ndarray, float], np.ndarray]

# Remove `*args` and the `if` inside the functions when removing deprecation
@deprecated_params(
    params="start_points, end_points, alpha",
    since="v0.14",
    until="v0.15",
    message="Straight path is now returning interpolating function to make it consistent with other path functions. Use straight_path()(a,b,c) instead of straight_path(a,b,c).",
)
def straight_path(*args) -> PATH_FUNC_TYPE:
    """Simplest path function. Each point in a set goes in a straight path toward its destination.

    Examples
    -------

    .. manim :: StraightPathExample

    class StraightPathExample(Scene):
        def construct(self):
            colors = [RED, GREEN, BLUE]

            starting_points = VGroup(
                *[Dot(LEFT + pos, color=color)
                  for pos, color in zip([UP, DOWN, LEFT], colors)]
            )

            finish_points = VGroup(
                *[Dot(RIGHT + pos, color=color)
                  for pos, color in zip([ORIGIN, UP, DOWN], colors)]
            )

            self.add(starting_points)
            self.add(finish_points)
            for dot in starting_points:
                self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

            self.wait()
            self.play(
                Transform(
                    starting_points,
                    finish_points,
                    path_func=utils.paths.straight_path(),
                    run_time=2,
                )
            )
            self.wait()

        """
        if len(args) > 0:
            return interpolate(*args)
        return interpolate

def path_along_circles(
    arc_angle: float, circles_centers: np.ndarray, axis: np.ndarray = OUT
) -> PATH_FUNC_TYPE:
    """This function transforms each point by moving it roughly along a circle, each with its own specified center.

    The path may be seen as each point smoothly changing its orbit from its starting position to its destination.

    Parameters
    -------

    arc_angle
        The angle each point traverses around the quasicircle.
    circles_centers
        The centers of each point's quasicircle to rotate around.
    axis
        The axis of rotation.

    Examples
    -------

    .. manim :: PathAlongCirclesExample

    class PathAlongCirclesExample(Scene):
        def construct(self):
            colors = [RED, GREEN, BLUE]

            starting_points = VGroup(
                *[Dot(LEFT + pos, color=color)
                  for pos, color in zip([UP, DOWN, LEFT], colors)]
            )

            finish_points = VGroup(
                *[Dot(RIGHT + pos, color=color)
                  for pos, color in zip([ORIGIN, UP, DOWN], colors)]
            )

            self.add(starting_points)
            self.add(finish_points)
            for dot in starting_points:
                self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

            circle_center = Dot(3 * LEFT)
            self.add(circle_center)

            self.wait()
```

```

        self.play(
            Transform(
                starting_points,
                finish_points,
                path_func=utils.paths.path_along_circles(
                    2 * PI, circle_center.get_center()
                ),
                run_time=3,
            )
        )
        self.wait()

"""
if np.linalg.norm(axis) == 0:
    axis = OUT
unit_axis = axis / np.linalg.norm(axis)

def path(start_points: np.ndarray, end_points: np.ndarray, alpha: float):
    detransformed_end_points = circles_centers + np.dot(
        end_points - circles_centers, rotation_matrix(-arc_angle, unit_axis).T
    )
    rot_matrix = rotation_matrix(alpha * arc_angle, unit_axis)
    return circles_centers + np.dot(
        interpolate(start_points, detransformed_end_points, alpha)
        - circles_centers,
        rot_matrix.T,
    )

```

return path

```

def path_along_arc(arc_angle: float, axis: np.ndarray = OUT) -> PATH_FUNC_TYPE:
    """This function transforms each point by moving it along a circular arc.

Parameters
-----
arc_angle
    The angle each point traverses around a circular arc.
axis
    The axis of rotation.

Examples
-----
.. manim :: PathAlongArcExample

class PathAlongArcExample(Scene):
    def construct(self):
        colors = [RED, GREEN, BLUE]

        starting_points = VGroup(
            *[Dot(LEFT + pos, color=color)
              for pos, color in zip([UP, DOWN, LEFT], colors)]
        )

        finish_points = VGroup(
            *[Dot(RIGHT + pos, color=color)
              for pos, color in zip([ORIGIN, UP, DOWN], colors)]
        )

        self.add(starting_points)
        self.add(finish_points)
        for dot in starting_points:
            self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

        self.wait()
        self.play(
            Transform(
                starting_points,
                finish_points,
                path_func=utils.paths.path_along_arc(TAU * 2 / 3),
                run_time=3,
            )
        )
        self.wait()

    """
    if abs(arc_angle) < STRAIGHT_PATH_THRESHOLD:
        return straight_path()
    if np.linalg.norm(axis) == 0:
        axis = OUT
    unit_axis = axis / np.linalg.norm(axis)

    def path(start_points: np.ndarray, end_points: np.ndarray, alpha: float):
        vект = end_points - start_points
        centers = start_points + 0.5 * vект
        if arc_angle != np.pi:
            centers += np.cross(unit_axis, vект / 2.0) / np.tan(arc_angle / 2)
        rot_matrix = rotation_matrix(alpha * arc_angle, unit_axis)
        return centers + np.dot(start_points - centers, rot_matrix.T)

    return path

def clockwise_path() -> PATH_FUNC_TYPE:
    """This function transforms each point by moving clockwise around a half circle.

Examples
-----
.. manim :: ClockwisePathExample

class ClockwisePathExample(Scene):
    def construct(self):
        colors = [RED, GREEN, BLUE]

        starting_points = VGroup(
            *[Dot(LEFT + pos, color=color)
              for pos, color in zip([UP, DOWN, LEFT], colors)]
        )

        finish_points = VGroup(
            *[Dot(RIGHT + pos, color=color)
              for pos, color in zip([ORIGIN, UP, DOWN], colors)]
        )

        self.add(starting_points)
        self.add(finish_points)
        for dot in starting_points:
            self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

        self.wait()
        self.play(
            Transform(
                starting_points,
                finish_points,
                path_func=utils.paths.clockwise_path(),
                run_time=3,
            )
        )
        self.wait()

```

```

        run_time=2,
    )
)
self.wait()

"""
return path_along_arc(-np.pi)

def counterclockwise_path() -> PATH_FUNC_TYPE:
    """This function transforms each point by moving counterclockwise around a half circle.

Examples
-----
.. manim :: CounterclockwisePathExample

class CounterclockwisePathExample(Scene):
    def construct(self):
        colors = [RED, GREEN, BLUE]

        starting_points = VGroup(
            *[Dot(LEFT + pos, color=color)
              for pos, color in zip([UP, DOWN, LEFT], colors)]
        )

        finish_points = VGroup(
            *[Dot(RIGHT + pos, color=color)
              for pos, color in zip([ORIGIN, UP, DOWN], colors)]
        )

        self.add(starting_points)
        self.add(finish_points)
        for dot in starting_points:
            self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

        self.wait()
        self.play(
            Transform(
                starting_points,
                finish_points,
                path_func=utils.paths.counterclockwise_path(),
                run_time=2,
            )
        )
        self.wait()

"""
return path_along_arc(np.pi)

def spiral_path(angle: float, axis: np.ndarray = OUT) -> PATH_FUNC_TYPE:
    """This function transforms each point by moving along a spiral to its destination.

Parameters
-----
angle
    The angle each point traverses around a spiral.
axis
    The axis of rotation.

Examples
-----
.. manim :: SpiralPathExample

class SpiralPathExample(Scene):
    def construct(self):
        colors = [RED, GREEN, BLUE]

        starting_points = VGroup(
            *[Dot(LEFT + pos, color=color)
              for pos, color in zip([UP, DOWN, LEFT], colors)]
        )

        finish_points = VGroup(
            *[Dot(RIGHT + pos, color=color)
              for pos, color in zip([ORIGIN, UP, DOWN], colors)]
        )

        self.add(starting_points)
        self.add(finish_points)
        for dot in starting_points:
            self.add(TracedPath(dot.get_center, stroke_color=dot.get_color()))

        self.wait()
        self.play(
            Transform(
                starting_points,
                finish_points,
                path_func=utils.paths.spiral_path(2 * TAU),
                run_time=5,
            )
        )
        self.wait()

"""
if abs(angle) < STRAIGHT_PATH_THRESHOLD:
    return straight_path()
if np.linalg.norm(axis) == 0:
    axis = OUT
unit_axis = axis / np.linalg.norm(axis)

def path(start_points: np.ndarray, end_points: np.ndarray, alpha: float):
    rot_matrix = rotation_matrix((alpha - 1) * angle, unit_axis)
    return start_points + alpha * np.dot(end_points - start_points, rot_matrix.T)

return path

```

logger_utils.py

```
"""Utilities to create and set the logger.

Manim's logger can be accessed as ``manim.logger``, or as
``logging.getLogger("manim")``, once the library has been imported. Manim also
exports a second object, ``console``, which should be used to print on screen
messages that need not be logged.

Both ``logger`` and ``console`` use the ``rich`` library to produce rich text
format.

"""

from __future__ import annotations

import configparser
import copy
import json
import logging
import sys
from typing import TYPE_CHECKING

from rich import color, errors
from rich import print as printf
from rich.console import Console
from rich.logging import RichHandler
from rich.theme import Theme

if TYPE_CHECKING:
    from pathlib import Path

HIGHLIGHTED_KEYWORDS = [ # these keywords are highlighted specially
    "Played",
    "animations",
    "scene",
    "Reading",
    "Writing",
    "script",
    "arguments",
    "Invalid",
    "Aborting",
    "module",
    "File",
    "Rendering",
    "Rendered",
]

WRONG_COLOR_CONFIG_MSG = """
[logging.level.error]Your colour configuration couldn't be parsed.
Loading the default color configuration.[/logging.level.error]
"""

def make_logger(
    parser: configparser.ConfigParser,
    verbosity: str,
) -> tuple[logging.Logger, Console]:
    """Make the manim logger and console.

    Parameters
    -----
    parser
        A parser containing any .cfg files in use.

    verbosity
        The verbosity level of the logger.

    Returns
    -----
    :class:`logging.Logger`, :class:`rich.Console`, :class:`rich.Console`
        The manim logger and consoles. The first console outputs
        to stdout, the second to stderr. All use the theme returned by
        :func:`parse_theme`.

    See Also
    -----
    :func:`~._config.utils.make_config_parser`, :func:`parse_theme`

    Notes
    -----
    The ``parser`` is assumed to contain only the options related to
    configuring the logger at the top level.

    """

    from rich import color, errors
    from rich.console import Console
    from rich.logging import RichHandler
    from rich.theme import Theme

    from . import parse_theme

    logger = logging.getLogger("manim")
    logger.setLevel(verbosity)

    if parser:
        config = configparser.ConfigParser()
        config.read_file(parser)
        theme = parse_theme(config)
        logger.setLevel(theme.log_level)

        logger.info("Using configuration file %s", parser.filename)
        logger.info("Using theme %s", theme.name)

        logger.info("Color configuration loaded successfully")

    else:
        logger.info("No configuration file provided, using defaults")
```

```

# Throughout the codebase, use console.print() instead of print()
theme = parse_theme(parser)
console = Console(theme=theme)

# With rich 9.5.0+ we could pass stderr=True instead
error_console = Console(theme=theme, file=sys.stderr)

# set the rich handler
RichHandler.KEYWORDS = HIGHLIGHTED_KEYWORDS
rich_handler = RichHandler(
    console=console,
    show_time=parser.getboolean("log_timestamps"),
)

# finally, the logger
logger = logging.getLogger("manim")
logger.addHandler(rich_handler)
logger.setLevel(verbosity)

return logger, console, error_console

def parse_theme(parser: configparser.ConfigParser) -> Theme:
    """Configure the rich style of logger and console output.

    Parameters
    -----
    parser
        A parser containing any .cfg files in use.

    Returns
    -----
    :class:`rich.Theme`
        The rich theme to be used by the manim logger.

    See Also
    -----
    :func:`make_logger`.

    """
    theme = {key.replace("_", ".") : parser[key] for key in parser}

    theme["log.width"] = None if theme["log.width"] == "-1" else int(theme["log.width"])
    theme["log.height"] = (
        None if theme["log.height"] == "-1" else int(theme["log.height"]))
    theme["log.timestamps"] = False
    try:
        custom_theme = Theme(
            {
                k: v
                for k, v in theme.items()
                if k not in ["log.width", "log.height", "log.timestamps"]
            },
        )
    except (color.ColorParseError, errors.StyleSyntaxError):
        printf(WRONG_COLOR_CONFIG_MSG)
        custom_theme = None

    return custom_theme

def set_file_logger(scene_name: str, module_name: str, log_dir: Path) -> None:
    """Add a file handler to manim logger.

    The path to the file is built using ``config.log_dir``.

    Parameters
    -----
    scene_name
        The name of the scene, used in the name of the log file.
    module_name
        The name of the module, used in the name of the log file.
    log_dir
        Path to the folder where log files are stored.
    """
    # Note: The log file name will be
    # <name_of_animation_file>_<name_of_scene>.log, gotten from config. So it
    # can differ from the real name of the scene. <name_of_scene> would only
    # appear if scene name was provided when manim was called.
    log_file_name = f"{module_name}_{scene_name}.log"
    log_file_path = log_dir / log_file_name

    file_handler = logging.FileHandler(log_file_path, mode="w")
    file_handler.setFormatter(JSONFormatter())

    logger = logging.getLogger("manim")

```

```
logger.addHandler(file_handler)
logger.info("Log file will be saved in %(logpath)s", {"logpath": log_file_path})

class JSONFormatter(logging.Formatter):
    """A formatter that outputs logs in a custom JSON format.

    This class is used internally for testing purposes.

    """

    def format(self, record: dict) -> str:
        """Format the record in a custom JSON format."""
        record_c = copy.deepcopy(record)
        if record_c.args:
            for arg in record_c.args:
                record_c.args[arg] = "<>"
        return json.dumps(
            {
                "levelname": record_c.levelname,
                "module": record_c.module,
                "message": super().format(record_c),
            },
        )
```

graph.py

```
"""Mobjects used to represent mathematical graphs (think graph theory, not plotting)."""

from __future__ import annotations

__all__ = [
    "Graph",
    "DiGraph",
]
import itertools as it
from copy import copy
from typing import Hashable, Iterable

import networkx as nx
import numpy as np

from manim.animation.composition import AnimationGroup
from manim.animation.creation import Create, Uncreate
from manim.mobject.geometry.arc import Dot, LabeledDot
from manim.mobject.geometry.line import Line
from manim.mobject import Mobject, override_animate
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl.opengl_mobject import OpenGLMobject
from manim.mobject.text.tex_mobject import MathTex
from manim.mobject.types.vectorized_mobject import VMobject
from manim.utils.color import BLACK

def _determine_graph_layout(
    nx_graph: nx.classes.graph.Graph | nx.classes.digraph.DiGraph,
    layout: str | dict = "spring",
    layout_scale: float = 2,
    layout_config: dict | None = None,
    partitions: list[list[Hashable]] | None = None,
    root_vertex: Hashable | None = None,
) -> dict:
    automatic_layouts = {
        "circular": nx.layout.circular_layout,
        "kamada_kawai": nx.layout.kamada_kawai_layout,
        "planar": nx.layout.planar_layout,
        "random": nx.layout.random_layout,
        "shell": nx.layout.shell_layout,
        "spectral": nx.layout.spectral_layout,
        "partite": nx.layout.multipartite_layout,
        "tree": _tree_layout,
        "spiral": nx.layout.spiral_layout,
        "spring": nx.layout.spring_layout,
    }
    custom_layouts = ["random", "partite", "tree"]

    if layout_config is None:
        layout_config = {}

    if isinstance(layout, dict):
        return layout
    elif layout in automatic_layouts and layout not in custom_layouts:
        auto_layout = automatic_layouts[layout](
            nx_graph, scale=layout_scale, **layout_config
        )
        # NetworkX returns a dictionary of 3D points if the dimension
        # is specified to be 3. Otherwise, it returns a dictionary of
        # 2D points, so adjusting is required.
        if layout_config.get("dim") == 3:
            return auto_layout
        else:
            return {k: np.append(v, [0]) for k, v in auto_layout.items()}
    elif layout == "tree":
        return _tree_layout(
            nx_graph, root_vertex=root_vertex, scale=layout_scale, **layout_config
        )
    elif layout == "partite":
        if partitions is None or len(partitions) == 0:
            raise ValueError(
                "The partite layout requires the 'partitions' parameter to contain the partition of the vertices",
            )
        partition_count = len(partitions)
        for i in range(partition_count):
            for v in partitions[i]:
                if nx_graph.nodes[v] is None:
                    raise ValueError(
                        "The partition must contain arrays of vertices in the graph",
                    )
                nx_graph.nodes[v]["subset"] = i
        # Add missing vertices to their own side
        for v in nx_graph.nodes:
            if "subset" not in nx_graph.nodes[v]:
                nx_graph.nodes[v]["subset"] = partition_count
        auto_layout = automatic_layouts["partite"](
            nx_graph, scale=layout_scale, **layout_config
        )
        return {k: np.append(v, [0]) for k, v in auto_layout.items()}
    elif layout == "random":
        # the random layout places coordinates in [0, 1)
```

```

# we need to rescale manually afterwards...
auto_layout = automatic_layouts["random"](nx_graph, **layout_config)
for k, v in auto_layout.items():
    auto_layout[k] = 2 * layout_scale * (v - np.array([0.5, 0.5]))
return {k: np.append(v, [0]) for k, v in auto_layout.items()}
else:
    raise ValueError(
        f"The layout '{layout}' is neither a recognized automatic layout, "
        "nor a vertex placement dictionary.",
    )

def _tree_layout(
    T: nx.classes.graph.Graph | nx.classes.digraph.DiGraph,
    root_vertex: Hashable | None,
    scale: float | tuple | None = 2,
    vertex_spacing: tuple | None = None,
    orientation: str = "down",
):
    children = {root_vertex: list(T.neighbors(root_vertex))}

    if not nx.is_tree(T):
        raise ValueError("The tree layout must be used with trees")
    if root_vertex is None:
        raise ValueError("The tree layout requires the root_vertex parameter")

    # The following code is SageMath's tree layout implementation, taken from
    # https://github.com/sagemath/sage/blob/cc60cfefc4576fed8b01f0fc487271bdee3cefed/src/graphs/graph_plot.py#L1447

    # Always make a copy of the children because they get eaten
    stack = [list(children[root_vertex]).copy()]
    stick = [root_vertex]
    parent = {u: root_vertex for u in children[root_vertex]}
    pos = {}
    obstruction = [0.0] * len(T)
    if orientation == "down":
        o = -1
    else:
        o = 1

    def slide(v, dx):
        """
        Shift the vertex v and its descendants to the right by dx.
        Precondition: v and its descendants have already had their
        positions computed.
        """
        level = [v]
        while level:
            nextlevel = []
            for u in level:
                x, y = pos[u]
                x += dx
                obstruction[y] = max(x + 1, obstruction[y])
                pos[u] = x, y
                nextlevel += children[u]
            level = nextlevel

        while stack:
            C = stack[-1]
            if not C:
                p = stick.pop()
                stack.pop()
                cp = children[p]
                y = o * len(stack)
                if not cp:
                    x = obstruction[y]
                    pos[p] = x, y
                else:
                    x = sum(pos[c][0] for c in cp) / float(len(cp))
                    pos[p] = x, y
                    ox = obstruction[y]
                    if x < ox:
                        slide(p, ox - x)
                    x = ox
                obstruction[y] = x + 1
                continue

            t = C.pop()
            pt = parent[t]

            ct = [u for u in list(T.neighbors(t)) if u != pt]
            for c in ct:
                parent[c] = t
            children[t] = copy(ct)

            stack.append(ct)
            stick.append(t)

    # the resulting layout is then rescaled again to fit on Manim's canvas

    x_min = min(pos.values(), key=lambda t: t[0])[0]
    x_max = max(pos.values(), key=lambda t: t[0])[0]
    y_min = min(pos.values(), key=lambda t: t[1])[1]
    y_max = max(pos.values(), key=lambda t: t[1])[1]
    center = np.array([x_min + x_max, y_min + y_max, 0]) / 2
    height = y_max - y_min
    width = x_max - x_min
    if vertex_spacing is None:
        if isinstance(scale, (float, int)) and (width > 0 or height > 0):
            sf = 2 * scale / max(width, height)
        elif isinstance(scale, tuple):

```

```

if scale[0] is not None and width > 0:
    sw = 2 * scale[0] / width
else:
    sw = 1

if scale[1] is not None and height > 0:
    sh = 2 * scale[1] / height
else:
    sh = 1

sf = np.array([sw, sh, 0])
else:
    sf = 1
else:
    sx, sy = vertex_spacing
    sf = np.array([sx, sy, 0])
return {v: (np.array([x, y, 0]) - center) * sf for v, (x, y) in pos.items()}

class GenericGraph(VMobject, metaclass=ConvertToOpenGL):
    """Abstract base class for graphs (that is, a collection of vertices connected with edges).

    Graphs can be instantiated by passing both a list of (distinct, hashable) vertex names, together with list of edges (as tuples of vertex names). See the examples for concrete implementations of this class for details.

    .. note::

        This implementation uses updaters to make the edges move with the vertices.

    See also
    -----
    :class:`.Graph`
    :class:`.DiGraph`


    Parameters
    -----
    vertices
        A list of vertices. Must be hashable elements.
    edges
        A list of edges, specified as tuples ``(u, v)`` where both ``u`` and ``v`` are vertices.
    labels
        Controls whether or not vertices are labeled. If ``False`` (the default), the vertices are not labeled; if ``True`` they are labeled using their names (as specified in ``vertices``) via :class:`~.MathTex`. Alternatively, custom labels can be specified by passing a dictionary whose keys are the vertices, and whose values are the corresponding vertex labels (rendered via, e.g., :class:`~.Text` or :class:`~.Tex`).
    label_fill_color
        Sets the fill color of the default labels generated when ``labels`` is set to ``True``. Has no effect for other values of ``labels``.
    layout
        Either one of ``"spring"`` (the default), ``"circular"`` , ``"kamada_kawai"`` , ``"planar"`` , ``"random"`` , ``"shell"`` , ``"spectral"`` , ``"spiral"`` , ``"tree"`` , and ``"partite"`` for automatic vertex positioning using ``networkx`` (see `their documentation <https://networkx.org/documentation/stable/reference/drawing.html#module-networkx.drawing.layout>`_ for more details), or a dictionary specifying a coordinate (value) for each vertex (key) for manual positioning.
    layout_config
        Only for automatically generated layouts. A dictionary whose entries are passed as keyword arguments to the automatic layout algorithm specified via ``layout`` of ``networkx``. The ``tree`` layout also accepts a special parameter ``vertex_spacing`` passed as a keyword argument inside the ``layout_config`` dictionary. Passing a tuple ``(space_x, space_y)`` as this argument overrides the value of ``layout_scale`` and ensures that vertices are arranged in a way such that the centers of siblings in the same layer are at least ``space_x`` units apart horizontally, and neighboring layers are spaced ``space_y`` units vertically.
    layout_scale
        The scale of automatically generated layouts: the vertices will be arranged such that the coordinates are located within the interval ``[-scale, scale]``. Some layouts accept a tuple ``(scale_x, scale_y)`` causing the first coordinate to be in the interval ``[-scale_x, scale_x]`` , and the second in ``[-scale_y, scale_y]``. Default: 2.
    vertex_type
        The mobject class used for displaying vertices in the scene.
    vertex_config
        Either a dictionary containing keyword arguments to be passed to the class specified via ``vertex_type`` , or a dictionary whose keys are the vertices, and whose values are dictionaries containing keyword arguments for the mobject related to the corresponding vertex.
    vertex_mobjects
        A dictionary whose keys are the vertices, and whose values are mobjects to be used as vertices. Passing vertices here overrides all other configuration options for a vertex.
    edge_type
        The mobject class used for displaying edges in the scene.
    edge_config
        Either a dictionary containing keyword arguments to be passed to the class specified via ``edge_type`` , or a dictionary whose keys are the edges, and whose values are dictionaries containing keyword arguments for the mobject related to the corresponding edge.
"""

```

```

def __init__(
    self,
    vertices: list[Hashable],
    edges: list[tuple[Hashable, Hashable]],
    labels: bool | dict = False,
    label_fill_color: str = BLACK,
    layout: str | dict = "spring",
    layout_scale: float | tuple = 2,
    layout_config: dict | None = None,
    vertex_type: type[Mobject] = Dot,
    vertex_config: dict | None = None,
    vertex_mobjects: dict | None = None,
    edge_type: type[Mobject] = Line,
    partitions: list[list[Hashable]] | None = None,
    root_vertex: Hashable | None = None,
    edge_config: dict | None = None,
) -> None:
    super().__init__()

    nx_graph = self._empty_networkx_graph()
    nx_graph.add_nodes_from(vertices)
    nx_graph.add_edges_from(edges)
    self._graph = nx_graph

    self._layout = _determine_graph_layout(
        nx_graph,
        layout=layout,
        layout_scale=layout_scale,
        layout_config=layout_config,
        partitions=partitions,
        root_vertex=root_vertex,
    )

    if isinstance(labels, dict):
        self._labels = labels
    elif isinstance(labels, bool):
        if labels:
            self._labels = {
                v: MathTex(v, fill_color=label_fill_color) for v in vertices
            }
        else:
            self._labels = {}

    if self._labels and vertex_type is Dot:
        vertex_type = LabeledDot

    if vertex_mobjects is None:
        vertex_mobjects = {}

    # build vertex_config
    if vertex_config is None:
        vertex_config = {}
    default_vertex_config = {}
    if vertex_config:
        default_vertex_config = {
            k: v for k, v in vertex_config.items() if k not in vertices
        }
    self._vertex_config = {
        v: vertex_config.get(v, copy(default_vertex_config)) for v in vertices
    }
    self.default_vertex_config = default_vertex_config
    for v, label in self._labels.items():
        self._vertex_config[v]["label"] = label

    self.vertices = {v: vertex_type(**self._vertex_config[v]) for v in vertices}
    self.vertices.update(vertex_mobjects)
    for v in self.vertices:
        self[v].move_to(self._layout[v])

    # build edge_config
    if edge_config is None:
        edge_config = {}
    default_tip_config = {}
    default_edge_config = {}
    if edge_config:
        default_tip_config = edge_config.pop("tip_config", {})
        default_edge_config = {
            k: v
            for k, v in edge_config.items()
            if not isinstance(
                k, tuple
            ) # everything that is not an edge is an option
        }
    self._edge_config = {}
    self._tip_config = {}
    for e in edges:
        if e in edge_config:
            self._tip_config[e] = edge_config[e].pop(
                "tip_config", copy(default_tip_config)
            )
            self._edge_config[e] = edge_config[e]
        else:
            self._tip_config[e] = copy(default_tip_config)
            self._edge_config[e] = copy(default_edge_config)

    self.default_edge_config = default_edge_config
    self._populate_edge_dict(edges, edge_type)

    self.add(*self.vertices.values())
    self.add(*self.edges.values())

```

```

    self.add_updater(self.update_edges)

@staticmethod
def _empty_networkx_graph():
    """Return an empty networkx graph for the given graph type."""
    raise NotImplementedError("To be implemented in concrete subclasses")

def _populate_edge_dict(
    self, edges: list[tuple[Hashable, Hashable]], edge_type: type[Mobject]
):
    """Helper method for populating the edges of the graph."""
    raise NotImplementedError("To be implemented in concrete subclasses")

def __getitem__(self: Graph, v: Hashable) -> Mobject:
    return self.vertices[v]

def _create_vertex(
    self,
    vertex: Hashable,
    position: np.ndarray | None = None,
    label: bool = False,
    label_fill_color: str = BLACK,
    vertex_type: type[Mobject] = Dot,
    vertex_config: dict | None = None,
    vertex_mobject: dict | None = None,
) -> tuple[Hashable, np.ndarray, dict, Mobject]:
    if position is None:
        position = self.get_center()

    if vertex_config is None:
        vertex_config = {}

    if vertex in self.vertices:
        raise ValueError(
            f"Vertex identifier '{vertex}' is already used for a vertex in this graph."
        )

    if label is True:
        label = MathTex(vertex, fill_color=label_fill_color)
    elif vertex in self._labels:
        label = self._labels[vertex]
    elif not isinstance(label, (Mobject, OpenGLMobject)):
        label = None

    base_vertex_config = copy(self.default_vertex_config)
    base_vertex_config.update(vertex_config)
    vertex_config = base_vertex_config

    if label is not None:
        vertex_config["label"] = label
        if vertex_type is Dot:
            vertex_type = LabeledDot

    if vertex_mobject is None:
        vertex_mobject = vertex_type(**vertex_config)

    vertex_mobject.move_to(position)

    return (vertex, position, vertex_config, vertex_mobject)

def _add_created_vertex(
    self,
    vertex: Hashable,
    position: np.ndarray,
    vertex_config: dict,
    vertex_mobject: Mobject,
) -> Mobject:
    if vertex in self.vertices:
        raise ValueError(
            f"Vertex identifier '{vertex}' is already used for a vertex in this graph."
        )

    self._graph.add_node(vertex)
    self._layout[vertex] = position

    if "label" in vertex_config:
        self._labels[vertex] = vertex_config["label"]

    self._vertex_config[vertex] = vertex_config

    self.vertices[vertex] = vertex_mobject
    self.vertices[vertex].move_to(position)
    self.add(self.vertices[vertex])

    return self.vertices[vertex]

def _add_vertex(
    self,
    vertex: Hashable,
    position: np.ndarray | None = None,
    label: bool = False,
    label_fill_color: str = BLACK,
    vertex_type: type[Mobject] = Dot,
    vertex_config: dict | None = None,
    vertex_mobject: dict | None = None,
) -> Mobject:
    """Add a vertex to the graph.

    Parameters
    -----

```

```

vertex
    A hashable vertex identifier.
position
    The coordinates where the new vertex should be added. If ``None``, the center
    of the graph is used.
label
    Controls whether or not the vertex is labeled. If ``False`` (the default),
    the vertex is not labeled; if ``True`` it is labeled using its
    names (as specified in ``vertex``) via :class:`~.MathTex`. Alternatively,
    any :class:`~.Mobject` can be passed to be used as the label.
label_fill_color
    Sets the fill color of the default labels generated when ``labels``
    is set to ``True``. Has no effect for other values of ``label``.
vertex_type
    The mobject class used for displaying vertices in the scene.
vertex_config
    A dictionary containing keyword arguments to be passed to
    the class specified via ``vertex_type``.
vertex_mobject
    The mobject to be used as the vertex. Overrides all other
    vertex customization options.
"""
return self._add_created_vertex(
    *self._create_vertex(
        vertex=vertex,
        position=position,
        label=label,
        label_fill_color=label_fill_color,
        vertex_type=vertex_type,
        vertex_config=vertex_config,
        vertex_mobject=vertex_mobject,
    )
)
def _create_vertices(
    self: Graph,
    *vertices: Hashable,
    positions: dict | None = None,
    labels: bool = False,
    label_fill_color: str = BLACK,
    vertex_type: type[Mobject] = Dot,
    vertex_config: dict | None = None,
    vertex_mobjects: dict | None = None,
) -> Iterable[tuple[Hashable, np.ndarray, dict, Mobject]]:
    if positions is None:
        positions = {}
    if vertex_mobjects is None:
        vertex_mobjects = {}

    graph_center = self.get_center()
    base_positions = {v: graph_center for v in vertices}
    base_positions.update(positions)
    positions = base_positions

    if isinstance(labels, bool):
        labels = {v: labels for v in vertices}
    else:
        assert isinstance(labels, dict)
        base_labels = {v: False for v in vertices}
        base_labels.update(labels)
        labels = base_labels

    if vertex_config is None:
        vertex_config = copy(self.default_vertex_config)

    assert isinstance(vertex_config, dict)
    base_vertex_config = copy(self.default_vertex_config)
    base_vertex_config.update(
        {key: val for key, val in vertex_config.items() if key not in vertices},
    )
    vertex_config = {
        v: (vertex_config[v] if v in vertex_config else copy(base_vertex_config))
        for v in vertices
    }

    return [
        self._create_vertex(
            v,
            position=positions[v],
            label=labels[v],
            label_fill_color=label_fill_color,
            vertex_type=vertex_type,
            vertex_config=vertex_config[v],
            vertex_mobject=vertex_mobjects[v] if v in vertex_mobjects else None,
        )
        for v in vertices
    ]
def add_vertices(
    self: Graph,
    *vertices: Hashable,
    positions: dict | None = None,
    labels: bool = False,
    label_fill_color: str = BLACK,
    vertex_type: type[Mobject] = Dot,
    vertex_config: dict | None = None,
    vertex_mobjects: dict | None = None,
):
    """Add a list of vertices to the graph.

```

```

Parameters
-----
vertices
    Hashable vertex identifiers.
positions
    A dictionary specifying the coordinates where the new vertices should be added.
    If ``None``, all vertices are created at the center of the graph.
labels
    Controls whether or not the vertex is labeled. If ``False`` (the default),
    the vertex is not labeled; if ``True`` it is labeled using its
    names (as specified in ``vertex``) via :class:`~.MathTex`. Alternatively,
    any :class:`~.Mobject` can be passed to be used as the label.
label_fill_color
    Sets the fill color of the default labels generated when ``labels``
    is set to ``True``. Has no effect for other values of ``labels``.
vertex_type
    The mobject class used for displaying vertices in the scene.
vertex_config
    A dictionary containing keyword arguments to be passed to
    the class specified via ``vertex_type``.
vertex_mobjects
    A dictionary whose keys are the vertex identifiers, and whose
    values are mobjects that should be used as vertices. Overrides
    all other vertex customization options.
"""
return [
    self._add_created_vertex(*v)
    for v in self._create_vertices(
        *vertices,
        positions=positions,
        labels=labels,
        label_fill_color=label_fill_color,
        vertex_type=vertex_type,
        vertex_config=vertex_config,
        vertex_mobjects=vertex_mobjects,
    )
]
]

@Override_animate(add_vertices)
def _add_vertices_animation(self, *args, anim_args=None, **kwargs):
    if anim_args is None:
        anim_args = {}

    animation = anim_args.pop("animation", Create)

    vertex_mobjects = self._create_vertices(*args, **kwargs)

    def on_finish(scene: Scene):
        for v in vertex_mobjects:
            scene.remove(v[-1])
            self._add_created_vertex(*v)

    return AnimationGroup(
        *(animation(v[-1], **anim_args) for v in vertex_mobjects),
        group=self,
        _on_finish=on_finish,
    )

def _remove_vertex(self, vertex):
    """Remove a vertex (as well as all incident edges) from the graph.

Parameters
-----
vertex
    The identifier of a vertex to be removed.

Returns
-----
Group
    A mobject containing all removed objects.

"""
if vertex not in self.vertices:
    raise ValueError(
        f"The graph does not contain a vertex with identifier '{vertex}'",
    )

self._graph.remove_node(vertex)
self._layout.pop(vertex)
if vertex in self._labels:
    self._labels.pop(vertex)
self._vertex_config.pop(vertex)

edge_tuples = [e for e in self.edges if vertex in e]
for e in edge_tuples:
    self._edge_config.pop(e)
to_remove = [self.edges.pop(e) for e in edge_tuples]
to_remove.append(self.vertices.pop(vertex))

self.remove(*to_remove)
return self.get_group_class()(*to_remove)

def remove_vertices(self, *vertices):
    """Remove several vertices from the graph.

Parameters
-----

```

```

vertices
    Vertices to be removed from the graph.

Examples
-----
::

>>> G = Graph([1, 2, 3], [(1, 2), (2, 3)])
>>> removed = G.remove_vertices(2, 3); removed
VGroup(Line, Line, Dot, Dot)
>>> G
Undirected graph on 1 vertices and 0 edges

"""
mobjects = []
for v in vertices:
    mobjects.extend(self._remove_vertex(v).submobjects)
return self.get_group_class()(*mobjects)

@Override_animate(remove_vertices)
def _remove_vertices_animation(self, *vertices, anim_args=None):
    if anim_args is None:
        anim_args = {}

    animation = anim_args.pop("animation", Uncreate)

    mobjects = self.remove_vertices(*vertices)
    return AnimationGroup(
        *(animation(mobj, **anim_args) for mobj in mobjects), group=self
    )

def _add_edge(
    self,
    edge: tuple[Hashable, Hashable],
    edge_type: type[Mobject] = Line,
    edge_config: dict | None = None,
):
    """Add a new edge to the graph.

Parameters
-----
edge
    The edge (as a tuple of vertex identifiers) to be added. If a non-existing
    vertex is passed, a new vertex with default settings will be created. Create
    new vertices yourself beforehand to customize them.
edge_type
    The mobject class used for displaying edges in the scene.
edge_config
    A dictionary containing keyword arguments to be passed
    to the class specified via ``edge_type``.

Returns
-----
Group
    A group containing all newly added vertices and edges.

"""
if edge_config is None:
    edge_config = self.default_edge_config.copy()
added_mobjects = []
for v in edge:
    if v not in self.vertices:
        added_mobjects.append(self._add_vertex(v))
u, v = edge

self._graph.add_edge(u, v)

base_edge_config = self.default_edge_config.copy()
base_edge_config.update(edge_config)
edge_config = base_edge_config
self._edge_config[(u, v)] = edge_config

edge_mobject = edge_type(
    self[u].get_center(), self[v].get_center(), z_index=-1, **edge_config
)
self.edges[(u, v)] = edge_mobject

self.add(edge_mobject)
added_mobjects.append(edge_mobject)
return self.get_group_class()(*added_mobjects)

def add_edges(
    self,
    *edges: tuple[Hashable, Hashable],
    edge_type: type[Mobject] = Line,
    edge_config: dict | None = None,
    **kwargs,
):
    """Add new edges to the graph.

Parameters
-----
edges
    Edges (as tuples of vertex identifiers) to be added. If a non-existing
    vertex is passed, a new vertex with default settings will be created. Create
    new vertices yourself beforehand to customize them.
edge_type
    The mobject class used for displaying edges in the scene.
edge_config

```

```

A dictionary either containing keyword arguments to be passed
to the class specified via ``edge_type``, or a dictionary
whose keys are the edge tuples, and whose values are dictionaries
containing keyword arguments to be passed for the construction
of the corresponding edge.

kwargs
    Any further keyword arguments are passed to :meth:`.add_vertices`
    which is used to create new vertices in the passed edges.

Returns
-----
Group
    A group containing all newly added vertices and edges.

"""
if edge_config is None:
    edge_config = {}
non_edge_settings = {k: v for (k, v) in edge_config.items() if k not in edges}
base_edge_config = self.default_edge_config.copy()
base_edge_config.update(non_edge_settings)
base_edge_config = {e: base_edge_config.copy() for e in edges}
for e in edges:
    base_edge_config[e].update(edge_config.get(e, {}))
edge_config = base_edge_config

edge_vertices = set(it.chain(*edges))
new_vertices = [v for v in edge_vertices if v not in self.vertices]
added_vertices = self.add_vertices(*new_vertices, **kwargs)

added_mobjects = sum(
    (
        self._add_edge(
            edge,
            edge_type=edge_type,
            edge_config=edge_config[edge],
        ).submobjects
        for edge in edges
    ),
    added_vertices,
)
return self.get_group_class()(added_mobjects)

@Override_animate(add_edges)
def _add_edges_animation(self, *args, anim_args=None, **kwargs):
    if anim_args is None:
        anim_args = {}
    animation = anim_args.pop("animation", Create)

    mobjects = self.add_edges(*args, **kwargs)
    return AnimationGroup(
        *(animation(mobj, **anim_args) for mobj in mobjects), group=self
    )

def _remove_edge(self, edge: tuple[Hashable]):
    """Remove an edge from the graph.

Parameters
-----
edge
    The edge (i.e., a tuple of vertex identifiers) to be removed from the graph.

Returns
-----
Mobject
    The removed edge.

"""
if edge not in self.edges:
    raise ValueError(f"The graph does not contain a edge '{edge}'")

edge_mobject = self.edges.pop(edge)

self._graph.remove_edge(*edge)
self._edge_config.pop(edge, None)

self.remove(edge_mobject)
return edge_mobject

def remove_edges(self, *edges: tuple[Hashable]):
    """Remove several edges from the graph.

Parameters
-----
edges
    Edges to be removed from the graph.

Returns
-----
Group
    A group containing all removed edges.

"""
edge_mobjects = [self._remove_edge(edge) for edge in edges]
return self.get_group_class()(edge_mobjects)

@Override_animate(remove_edges)
def _remove_edges_animation(self, *edges, anim_args=None):
    if anim_args is None:
        anim_args = {}

```

```

animation = anim_args.pop("animation", Uncreate)

mobjects = self.remove_edges(*edges)
return AnimationGroup(*animation(mobj, **anim_args) for mobj in mobjects)

@classmethod
def from_networkx(
    cls, nxgraph: nx.classes.graph.Graph | nx.classes.digraph.DiGraph, **kwargs
):
    """Build a :class:`~.Graph` or :class:`~.DiGraph` from a
    given ``networkx`` graph.

    Parameters
    -----
    nxgraph
        A ``networkx`` graph or digraph.
    **kwargs
        Keywords to be passed to the constructor of :class:`~.Graph`.

    Examples
    -----
    .. manim:: ImportNetworkxGraph

        import networkx as nx

        nxgraph = nx.erdos_renyi_graph(14, 0.5)

        class ImportNetworkxGraph(Scene):
            def construct(self):
                G = Graph.from_networkx(nxgraph, layout="spring", layout_scale=3.5)
                self.play(Create(G))
                self.play(*[G[v].animate.move_to(5*RIGHT*np.cos(ind/7 * PI) +
                                                3*UP*np.sin(ind/7 * PI))
                           for ind, v in enumerate(G.vertices)])
                self.play(Uncreate(G))

    """
    return cls(list(nxgraph.nodes), list(nxgraph.edges), **kwargs)

def change_layout(
    self,
    layout: str | dict = "spring",
    layout_scale: float = 2,
    layout_config: dict | None = None,
    partitions: list[list[Hashable]] | None = None,
    root_vertex: Hashable | None = None,
) -> Graph:
    """Change the layout of this graph.

    See the documentation of :class:`~.Graph` for details about the
    keyword arguments.

    Examples
    -----
    .. manim:: ChangeGraphLayout

        class ChangeGraphLayout(Scene):
            def construct(self):
                G = Graph([1, 2, 3, 4, 5], [(1, 2), (2, 3), (3, 4), (4, 5)],
                          layout={1: [-2, 0, 0], 2: [-1, 0, 0], 3: [0, 0, 0],
                                  4: [1, 0, 0], 5: [2, 0, 0]})
                self.play(Create(G))
                self.play(G.animate.change_layout("circular"))
                self.wait()

    """
    self._layout = _determine_graph_layout(
        self._graph,
        layout=layout,
        layout_scale=layout_scale,
        layout_config=layout_config,
        partitions=partitions,
        root_vertex=root_vertex,
    )
    for v in self.vertices:
        self[v].move_to(self._layout[v])
    return self

class Graph(GenericGraph):
    """An undirected graph (vertices connected with edges).

    The graph comes with an updater which makes the edges stick to
    the vertices when moved around. See :class:`.DiGraph` for
    a version with directed edges.

    See also
    -----
    :class:`.GenericGraph`

    Parameters
    -----
    vertices
        A list of vertices. Must be hashable elements.
    edges
    """


```

```

A list of edges, specified as tuples ``(u, v)`` where both ``u``
and ``v`` are vertices. The vertex order is irrelevant.
labels
Controls whether or not vertices are labeled. If ``False`` (the default),
the vertices are not labeled; if ``True`` they are labeled using their
names (as specified in ``vertices``) via :class:`~.MathTex`. Alternatively,
custom labels can be specified by passing a dictionary whose keys are
the vertices, and whose values are the corresponding vertex labels
(rendered via, e.g., :class:`~.Text` or :class:`~.Tex`).
label_fill_color
Sets the fill color of the default labels generated when ``labels``
is set to ``True``. Has no effect for other values of ``labels``.
layout
Either one of ``"spring"`` (the default), ``"circular"`` , ``"kamada_kawai"`` ,
``"planar"`` , ``"random"`` , ``"shell"`` , ``"spectral"`` , ``"spiral"`` , ``"tree"`` , and ``"partite"`` .
for automatic vertex positioning using ``networkx``
(see `their documentation <https://networkx.org/documentation/stable/reference/drawing.html#module-networkx.drawing.layout>`_
for more details), or a dictionary specifying a coordinate (value)
for each vertex (key) for manual positioning.
layout_config
Only for automatically generated layouts. A dictionary whose entries
are passed as keyword arguments to the automatic layout algorithm
specified via ``layout`` of ``networkx``.
The ``tree`` layout also accepts a special parameter ``vertex_spacing``
passed as a keyword argument inside the ``layout_config`` dictionary.
Passing a tuple ``(space_x, space_y)`` as this argument overrides
the value of ``layout_scale`` and ensures that vertices are arranged
in a way such that the centers of siblings in the same layer are
at least ``space_x`` units apart horizontally, and neighboring layers
are spaced ``space_y`` units vertically.
layout_scale
The scale of automatically generated layouts: the vertices will
be arranged such that the coordinates are located within the
interval ``[-scale, scale]``. Some layouts accept a tuple ``(scale_x, scale_y)``
causing the first coordinate to be in the interval ``[-scale_x, scale_x]``,
and the second in ``[-scale_y, scale_y]``. Default: 2.
vertex_type
The mobject class used for displaying vertices in the scene.
vertex_config
Either a dictionary containing keyword arguments to be passed to
the class specified via ``vertex_type``, or a dictionary whose keys
are the vertices, and whose values are dictionaries containing keyword
arguments for the mobject related to the corresponding vertex.
vertex_mobjects
A dictionary whose keys are the vertices, and whose values are
mobjects to be used as vertices. Passing vertices here overrides
all other configuration options for a vertex.
edge_type
The mobject class used for displaying edges in the scene.
edge_config
Either a dictionary containing keyword arguments to be passed
to the class specified via ``edge_type``, or a dictionary whose
keys are the edges, and whose values are dictionaries containing
keyword arguments for the mobject related to the corresponding edge.

```

Examples

First, we create a small graph and demonstrate that the edges move
together with the vertices.

```
.. manim:: MovingVertices

class MovingVertices(Scene):
    def construct(self):
        vertices = [1, 2, 3, 4]
        edges = [(1, 2), (2, 3), (3, 4), (1, 3), (1, 4)]
        g = Graph(vertices, edges)
        self.play(Create(g))
        self.wait()
        self.play(g[1].animate.move_to([1, 1, 0]),
                  g[2].animate.move_to([-1, 1, 0]),
                  g[3].animate.move_to([1, -1, 0]),
                  g[4].animate.move_to([-1, -1, 0]))
        self.wait()
```

There are several automatic positioning algorithms to choose from:

```
.. manim:: GraphAutoPosition
:save_last_frame:

class GraphAutoPosition(Scene):
    def construct(self):
        vertices = [1, 2, 3, 4, 5, 6, 7, 8]
        edges = [(1, 7), (1, 8), (2, 3), (2, 4), (2, 5),
                  (2, 8), (3, 4), (6, 1), (6, 2),
                  (6, 3), (7, 2), (7, 4)]
        autolayouts = ["spring", "circular", "kamada_kawai",
                      "planar", "random", "shell",
                      "spectral", "spiral"]
        graphs = [Graph(vertices, edges, layout=lt).scale(0.5)
                  for lt in autolayouts]
        r1 = VGroup(*graphs[:3]).arrange()
        r2 = VGroup(*graphs[3:6]).arrange()
        r3 = VGroup(*graphs[6:]).arrange()
        self.add(VGroup(r1, r2, r3).arrange(direction=DOWN))
```

Vertices can also be positioned manually:

```
.. manim:: GraphManualPosition
```

```

:save_last_frame:

class GraphManualPosition(Scene):
    def construct(self):
        vertices = [1, 2, 3, 4]
        edges = [(1, 2), (2, 3), (3, 4), (4, 1)]
        lt = {1: [0, 0, 0], 2: [1, 1, 0], 3: [1, -1, 0], 4: [-1, 0, 0]}
        G = Graph(vertices, edges, layout=lt)
        self.add(G)

```

The vertices in graphs can be labeled, and configurations for vertices and edges can be modified both by default and for specific vertices and edges.

.. note::

In ``edge_config``, edges can be passed in both directions: if `` (u, v) `` is an edge in the graph, both `` (u, v) `` as well as `` (v, u) `` can be used as keys in the dictionary.

```

.. manim:: LabeledModifiedGraph
:save_last_frame:

```

```

class LabeledModifiedGraph(Scene):
    def construct(self):
        vertices = [1, 2, 3, 4, 5, 6, 7, 8]
        edges = [(1, 7), (1, 8), (2, 3), (2, 4), (2, 5),
                  (2, 8), (3, 4), (6, 1), (6, 2),
                  (6, 3), (7, 2), (7, 4)]
        g = Graph(vertices, edges, layout="circular", layout_scale=3,
                  labels=True, vertex_config={7: {"fill_color": RED}},
                  edge_config={(1, 7): {"stroke_color": RED},
                               (2, 7): {"stroke_color": RED},
                               (4, 7): {"stroke_color": RED}})
        self.add(g)

```

You can also lay out a partite graph on columns by specifying a list of the vertices on each side and choosing the partite layout.

.. note::

All vertices in your graph which are not listed in any of the partitions are collected in their own partition and rendered in the rightmost column.

```

.. manim:: PartiteGraph
:save_last_frame:

```

```

import networkx as nx

class PartiteGraph(Scene):
    def construct(self):
        G = nx.Graph()
        G.add_nodes_from([0, 1, 2, 3])
        G.add_edges_from([(0, 2), (0, 3), (1, 2)])
        graph = Graph(list(G.nodes), list(G.edges), layout="partite", partitions=[[0, 1]])
        self.play(Create(graph))

```

The representation of a linear artificial neural network is facilitated by the use of the partite layout and defining partitions for each layer.

```

.. manim:: LinearNN
:save_last_frame:

```

```

class LinearNN(Scene):
    def construct(self):
        edges = []
        partitions = []
        c = 0
        layers = [2, 3, 3, 2] # the number of neurons in each layer

        for i in layers:
            partitions.append(list(range(c + 1, c + i + 1)))
            c += i
        for i, v in enumerate(layers[1:]):
            last = sum(layers[:i+1])
            for j in range(v):
                for k in range(last - layers[i], last):
                    edges.append((k + 1, j + last + 1))

        vertices = np.arange(1, sum(layers) + 1)

        graph = Graph(
            vertices,
            edges,
            layout='partite',
            partitions=partitions,
            layout_scale=3,
            vertex_config={'radius': 0.20},
        )
        self.add(graph)

```

The custom tree layout can be used to show the graph by distance from the root vertex. You must pass the root vertex of the tree.

.. manim:: Tree

```

import networkx as nx

class Tree(Scene):
    def construct(self):

```

```

G = nx.Graph()
G.add_node("ROOT")

for i in range(5):
    G.add_node("Child_%i" % i)
    G.add_node("Grandchild_%i" % i)
    G.add_node("Greatgrandchild_%i" % i)

    G.add_edge("ROOT", "Child_%i" % i)
    G.add_edge("Child_%i" % i, "Grandchild_%i" % i)
    G.add_edge("Grandchild_%i" % i, "Greatgrandchild_%i" % i)

self.play(Create(
    Graph(list(G.nodes), list(G.edges), layout="tree", root_vertex="ROOT")))

```

The following code sample illustrates the use of the ``vertex_spacing`` layout parameter specific to the ``"tree"`` layout. As mentioned above, setting ``vertex_spacing`` overrides the specified value for ``layout_scale``, and as such it is harder to control the size of the mobject. However, we can adjust the captured frame and zoom out by using a :class:`.MovingCameraScene`::

```

class LargeTreeGeneration(MovingCameraScene):
    DEPTH = 4
    CHILDREN_PER_VERTEX = 3
    LAYOUT_CONFIG = {"vertex_spacing": (0.5, 1)}
    VERTEX_CONF = {"radius": 0.25, "color": BLUE_B, "fill_opacity": 1}

    def expand_vertex(self, g, vertex_id: str, depth: int):
        new_vertices = [f"{vertex_id}/{i}" for i in range(self.CHILDREN_PER_VERTEX)]
        new_edges = [(vertex_id, child_id) for child_id in new_vertices]
        g.add_edges(*new_edges,
                    vertex_config=self.VERTEX_CONF,
                    positions={
                        k: g.vertices[vertex_id].get_center() + 0.1 * DOWN for k in new_vertices
                    },
                )
        if depth < self.DEPTH:
            for child_id in new_vertices:
                self.expand_vertex(g, child_id, depth + 1)

    return g

    def construct(self):
        g = Graph(["ROOT"], [], vertex_config=self.VERTEX_CONF)
        g = self.expand_vertex(g, "ROOT", 1)
        self.add(g)

        self.play(
            g.animate.change_layout(
                "tree",
                root_vertex="ROOT",
                layout_config=self.LAYOUT_CONFIG,
            )
        )
        self.play(self.camera.auto_zoom(g, margin=1), run_time=0.5)
    """

```

```
@staticmethod
def _empty_networkx_graph() -> nx.Graph:
    return nx.Graph()
```

```

def _populate_edge_dict(
    self, edges: list[tuple[Hashable, Hashable]], edge_type: type[Mobject]
):
    self.edges = {
        (u, v): edge_type(
            self[u].get_center(),
            self[v].get_center(),
            z_index=-1,
            **self._edge_config[(u, v)],
        )
        for (u, v) in edges
    }

def update_edges(self, graph):
    for (u, v), edge in graph.edges.items():
        # Undirected graph has a Line edge
        edge.put_start_and_end_on(graph[u].get_center(), graph[v].get_center())

def __repr__(self: Graph) -> str:
    return f"Undirected graph on {len(self.vertices)} vertices and {len(self.edges)} edges"
```

```
class DiGraph(GenericGraph):
    """A directed graph.
```

```
.. note::
```

In contrast to undirected graphs, the order in which vertices in a given edge are specified is relevant here.

See also

:class:`.GenericGraph`

Parameters

```

vertices
    A list of vertices. Must be hashable elements.
edges
    A list of edges, specified as tuples ``(u, v)`` where both ``u``
    and ``v`` are vertices. The edge is directed from ``u`` to ``v``.
labels
    Controls whether or not vertices are labeled. If ``False`` (the default),
    the vertices are not labeled; if ``True`` they are labeled using their
    names (as specified in ``vertices``) via :class:`~.MathTex`. Alternatively,
    custom labels can be specified by passing a dictionary whose keys are
    the vertices, and whose values are the corresponding vertex labels
    (rendered via, e.g., :class:`~.Text` or :class:`~.Tex`).
label_fill_color
    Sets the fill color of the default labels generated when ``labels``
    is set to ``True``. Has no effect for other values of ``labels``.
layout
    Either one of ``"spring"`` (the default), ``"circular"``, ``"kamada_kawai"``,
    ``"planar"``, ``"random"``, ``"shell"``, ``"spectral"``, ``"spiral"``, ``"tree"``, and ``"partite"``.
    for automatic vertex positioning using `networkx`_
    (see `their documentation <https://networkx.org/documentation/stable/reference/drawing.html#module-networkx.drawing.layout>`_
    for more details), or a dictionary specifying a coordinate (value)
    for each vertex (key) for manual positioning.
layout_config
    Only for automatically generated layouts. A dictionary whose entries
    are passed as keyword arguments to the automatic layout algorithm
    specified via ``layout`` of `networkx`_.
    The ``tree`` layout also accepts a special parameter ``vertex_spacing``
    passed as a keyword argument inside the ``layout_config`` dictionary.
    Passing a tuple ``(space_x, space_y)`` as this argument overrides
    the value of ``layout_scale`` and ensures that vertices are arranged
    in a way such that the centers of siblings in the same layer are
    at least ``space_x`` units apart horizontally, and neighboring layers
    are spaced ``space_y`` units vertically.
layout_scale
    The scale of automatically generated layouts: the vertices will
    be arranged such that the coordinates are located within the
    interval ``[scale, scale]``. Some layouts accept a tuple ``(scale_x, scale_y)``
    causing the first coordinate to be in the interval ``[-scale_x, scale_x]``,
    and the second in ``[-scale_y, scale_y]``. Default: 2.
vertex_type
    The mobject class used for displaying vertices in the scene.
vertex_config
    Either a dictionary containing keyword arguments to be passed to
    the class specified via ``vertex_type``, or a dictionary whose keys
    are the vertices, and whose values are dictionaries containing keyword
    arguments for the mobject related to the corresponding vertex.
vertex_mobjects
    A dictionary whose keys are the vertices, and whose values are
    mobjects to be used as vertices. Passing vertices here overrides
    all other configuration options for a vertex.
edge_type
    The mobject class used for displaying edges in the scene.
edge_config
    Either a dictionary containing keyword arguments to be passed
    to the class specified via ``edge_type``, or a dictionary whose
    keys are the edges, and whose values are dictionaries containing
    keyword arguments for the mobject related to the corresponding edge.
    You can further customize the tip by adding a ``tip_config`` dictionary
    for global styling, or by adding the dict to a specific ``edge_config``.

```

Examples

```
.. manim:: MovingDiGraph

class MovingDiGraph(Scene):
    def construct(self):
        vertices = [1, 2, 3, 4]
        edges = [(1, 2), (2, 3), (3, 4), (1, 3), (1, 4)]

        g = DiGraph(vertices, edges)

        self.add(g)
        self.play(
            g[1].animate.move_to([1, 1, 1]),
            g[2].animate.move_to([-1, 1, 2]),
            g[3].animate.move_to([1, -1, -1]),
            g[4].animate.move_to([-1, -1, 0]),
        )
        self.wait()
```

You can customize the edges and arrow tips globally or locally.

```
.. manim:: CustomDiGraph

class CustomDiGraph(Scene):
    def construct(self):
        vertices = [i for i in range(5)]
        edges = [
            (0, 1),
            (1, 2),
            (3, 2),
            (3, 4),
        ]

        edge_config = {
            "stroke_width": 2,
            "tip_config": {
                "tip_shape": ArrowSquareTip,
                "tip_length": 0.15,
```

```

        },
        (3, 4): {
            "color": RED,
            "tip_config": {"tip_length": 0.25, "tip_width": 0.25}
        },
    }

    g = DiGraph(
        vertices,
        edges,
        labels=True,
        layout="circular",
        edge_config=edge_config,
    ).scale(1.4)

    self.play(Create(g))
    self.wait()

Since this implementation respects the labels boundary you can also use
it for an undirected moving graph with labels.

.. manim:: UndirectedMovingDiGraph

class UndirectedMovingDiGraph(Scene):
    def construct(self):
        vertices = [i for i in range(5)]
        edges = [
            (0, 1),
            (1, 2),
            (3, 2),
            (3, 4),
        ]

        edge_config = {
            "stroke_width": 2,
            "tip_config": {"tip_length": 0, "tip_width": 0},
            (3, 4): {"color": RED},
        }

        g = DiGraph(
            vertices,
            edges,
            labels=True,
            layout="circular",
            edge_config=edge_config,
        ).scale(1.4)

        self.play(Create(g))
        self.wait()

        self.play(
            g[1].animate.move_to([1, 1, 1]),
            g[2].animate.move_to([-1, 1, 2]),
            g[3].animate.move_to([-1.5, -1.5, -1]),
            g[4].animate.move_to([1, -2, -1]),
        )
        self.wait()

    """
    @staticmethod
    def _empty_networkx_graph() -> nx.DiGraph:
        return nx.DiGraph()

    def _populate_edge_dict(
        self, edges: list[tuple[Hashable, Hashable]], edge_type: type[Mobject]
    ):
        self.edges = {
            (u, v): edge_type(
                self[u],
                self[v],
                z_index=-1,
                **self._edge_config[(u, v)],
            )
            for (u, v) in edges
        }

        for (u, v), edge in self.edges.items():
            edge.add_tip(**self._tip_config[(u, v)])

    def update_edges(self, graph):
        """Updates the edges to stick at their corresponding vertices.

        Arrow tips need to be repositioned since otherwise they can be
        deformed.
        """
        for (u, v), edge in graph.edges.items():
            edge_type = type(edge)
            tip = edge.pop_tips()[0]
            new_edge = edge_type(self[u], self[v], **self._edge_config[(u, v)])
            edge.become(new_edge)
            edge.add_tip(tip)

    def __repr__(self: DiGraph) -> str:
        return f"Directed graph on {len(self.vertices)} vertices and {len(self.edges)} edges"

```

brace.py

```
"""Mobject representing curly braces."""

from __future__ import annotations

__all__ = ["Brace", "BraceLabel", "ArcBrace", "BraceText", "BraceBetweenPoints"]

from typing import Sequence

import numpy as np
import svgelements as se

from manim._config import config
from manim.mobject.geometry.arc import Arc
from manim.mobject.geometry.line import Line
from manim.mobject.mobject import Mobject
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.text.tex_mobject import MathTex, Tex

from ...animation.composition import AnimationGroup
from ...animation.fading import FadeIn
from ...animation.growing import GrowFromCenter
from ...constants import *
from ...mobject.types.vectorized_mobject import VMobject
from ...utils.color import BLACK
from ..svg.svg_mobject import VMobjectFromSVGPath


class Brace(VMobjectFromSVGPath):
    """Takes a mobject and draws a brace adjacent to it.

    Passing a direction vector determines the direction from which the
    brace is drawn. By default it is drawn from below.

    Parameters
    -----
    mobject
        The mobject adjacent to which the brace is placed.
    direction :
        The direction from which the brace faces the mobject.

    See Also
    -----
    :class:`BraceBetweenPoints`


    Examples
    -----
    .. manim:: BraceExample
       :save_last_frame:

       class BraceExample(Scene):
           def construct(self):
               s = Square()
               self.add(s)
               for i in np.linspace(0.1,1.0,4):
                   br = Brace(s, sharpness=i)
                   t = Text(f"sharpness= {i}").next_to(br, RIGHT)
                   self.add(t)
                   self.add(br)
               VGroup(*self.mobjects).arrange(DOWN, buff=0.2)

    """

    def __init__(self,
                 mobject: Mobject,
                 direction: Sequence[float] | None = DOWN,
                 buff=0.2,
                 sharpness=2,
                 stroke_width=0,
                 fill_opacity=1.0,
                 background_stroke_width=0,
                 background_stroke_color=BLACK,
                 **kwargs,
                 ):
        path_string_template = (
            "m0.01216 0c-0.01152 -0.01216 6.103e-4 -0.01216 0.01311v0.007762c0.06776 "
            "0.122 0.1799 0.1455 0.2307 0.1455h{0}c0.03046 3.899e-4 0.07964 0.00449 "
            "0.1246 0.02636 0.0537 0.02695 0.07418 0.05816 0.08648 0.07769 0.001562 "
            "0.002538 0.004539 0.002563 0.01098 0.002563 0.006444-2e-8 0.009421-2.47e-"
            "5 0.01098-0.002563 0.0123-0.01953 0.03278-0.05074 0.08648-0.07769 0.04491"
            "-0.02187 0.09409-0.02597 0.1246-0.02636h{0}c0.05077 0 0.1629-0.02346 "
        )
```

```

        "0.2307-0.1455v-0.007762c-1.78e-6 -0.0125-6.365e-4 -0.01311-0.01216-0.0131"
        "1-0.006444-3.919e-8 -0.009348 2.448e-5 -0.01091 0.002563-0.0123 0.01953-"
        "0.03278 0.05074-0.08648 0.07769-0.04491 0.02187-0.09416 0.02597-0.1246 "
        "0.02636h{1}c-0.04786 0-0.1502 0.02094-0.2185 0.1256-0.06833-0.1046-0.1706"
        "-0.1256-0.2185-0.1256h{1}c-0.03046-3.899e-4 -0.07972-0.004491-0.1246-0.02"
        "636-0.0537-0.02695-0.07418-0.05816-0.08648-0.07769-0.001562-0.002538-"
        "0.004467-0.002563-0.01091-0.002563z"
    )
    default_min_width = 0.90552

    selfbuff = buff

    angle = -np.arctan2(*direction[:2]) + np.pi
    mobobject.rotate(-angle, about_point=ORIGIN)
    left = mobobject.get_corner(DOWN + LEFT)
    right = mobobject.get_corner(DOWN + RIGHT)
    target_width = right[0] - left[0]
    linear_section_length = max(
        0,
        (target_width * sharpness - default_min_width) / 2,
    )

    path = se.Path(
        path_string_template.format(
            linear_section_length,
            -linear_section_length,
        )
    )

super().__init__(
    path_obj=path,
    stroke_width=stroke_width,
    fill_opacity=fill_opacity,
    background_stroke_width=background_stroke_width,
    background_stroke_color=background_stroke_color,
    **kwargs,
)
self.flip(RIGHT)
self.stretch_to_fit_width(target_width)
self.shift(left - self.get_corner(UP + LEFT) + selfbuff * DOWN)

for mob in mobobject, self:
    mob.rotate(angle, about_point=ORIGIN)

def put_at_tip(self, mob, use_next_to=True, **kwargs):
    if use_next_to:
        mob.next_to(self.get_tip(), np.round(self.get_direction()), **kwargs)
    else:
        mob.move_to(self.get_tip())
        buff = kwargs.get("buff", DEFAULT_MOBJECT_TO_MOBJECT_BUFFER)
        shift_distance = mob.width / 2.0 + buff
        mob.shift(self.get_direction() * shift_distance)
    return self

def get_text(self, *text, **kwargs):
    text_mob = Tex(*text)
    self.put_at_tip(text_mob, **kwargs)
    return text_mob

def get_tex(self, *tex, **kwargs):
    tex_mob = MathTex(*tex)
    self.put_at_tip(tex_mob, **kwargs)
    return tex_mob

def get_tip(self):
    # Returns the position of the seventh point in the path, which is the tip.
    if config["renderer"] == "opengl":
        return self.points[34]

    return self.points[28] # = 7*4

def get_direction(self):
    vect = self.get_tip() - self.get_center()
    return vect / np.linalg.norm(vect)

class BraceLabel(VMobject, metaclass=ConvertToOpenGL):
    """Create a brace with a label attached.

    Parameters
    -----
    obj
        The mobobject adjacent to which the brace is placed.
    text
        The label text.
    brace_direction
        The direction of the brace. By default ``DOWN``.

```

```

label_constructor
    A class or function used to construct a mobject representing
    the label. By default :class:`~.MathTex` .
font_size
    The font size of the label, passed to the ``label_constructor``.
buff
    The buffer between the mobject and the brace.
brace_config
    Arguments to be passed to :class:`.Brace` .
kwargs
    Additional arguments to be passed to :class:`~.VMOBJECT` .
"""

def __init__(self,
             obj: Mobject,
             text: str,
             brace_direction: np.ndarray = DOWN,
             label_constructor: type = MathTex,
             font_size: float = DEFAULT_FONT_SIZE,
             buff: float = 0.2,
             brace_config: dict | None = None,
             **kwargs,
             ):
    self.label_constructor = label_constructor
    super().__init__(**kwargs)

    self.brace_direction = brace_direction
    if brace_config is None:
        brace_config = {}
    self.brace = Brace(obj, brace_direction, buff, **brace_config)

    if isinstance(text, (tuple, list)):
        self.label = self.label_constructor(font_size=font_size, *text, **kwargs)
    else:
        self.label = self.label_constructor(str(text), font_size=font_size)

    self.brace.put_at_tip(self.label)
    self.add(self.brace, self.label)

def creation_anim(self, label_anim=FadeIn, brace_anim=GrowFromCenter):
    return AnimationGroup(brace_anim(self.brace), label_anim(self.label))

def shift_brace(self, obj, **kwargs):
    if isinstance(obj, list):
        obj = self.get_group_class()(*obj)
    self.brace = Brace(obj, self.brace_direction, **kwargs)
    self.brace.put_at_tip(self.label)
    return self

def change_label(self, *text, **kwargs):
    self.label = self.label_constructor(*text, **kwargs)

    self.brace.put_at_tip(self.label)
    return self

def change_brace_label(self, obj, *text, **kwargs):
    self.shift_brace(obj)
    self.change_label(*text, **kwargs)
    return self

class BraceText(BraceLabel):
    def __init__(self, obj, text, label_constructor=Tex, **kwargs):
        super().__init__(obj, text, label_constructor=label_constructor, **kwargs)

class BraceBetweenPoints(Brace):
    """Similar to Brace, but instead of taking a mobject it uses 2
    points to place the brace.

    A fitting direction for the brace is
    computed, but it still can be manually overridden.
    If the points go from left to right, the brace is drawn from below.
    Swapping the points places the brace on the opposite side.

    Parameters
    -----
    point_1 :
        The first point.
    point_2 :
        The second point.
    direction :
        The direction from which the brace faces towards the points.

    Examples
    -----
    """

```

```

.. manim:: BraceBPExample

class BraceBPExample(Scene):
    def construct(self):
        p1 = [0,0,0]
        p2 = [1,2,0]
        brace = BraceBetweenPoints(p1,p2)
        self.play(Create(NumberPlane()))
        self.play(Create(brace))
        self.wait(2)
"""

def __init__(
    self,
    point_1: Sequence[float] | None,
    point_2: Sequence[float] | None,
    direction: Sequence[float] | None = ORIGIN,
    **kwargs,
):
    if all(direction == ORIGIN):
        line_vector = np.array(point_2) - np.array(point_1)
        direction = np.array([line_vector[1], -line_vector[0], 0])
    super().__init__(Line(point_1, point_2), direction=direction, **kwargs)

class ArcBrace(Brace):
    """Creates a :class:`~Brace` that wraps around an :class:`~.Arc`."""

    The direction parameter allows the brace to be applied
    from outside or inside the arc.

    .. warning::
        The :class:`ArcBrace` is smaller for arcs with smaller radii.

    .. note::
        The :class:`ArcBrace` is initially a vertical :class:`Brace` defined by the
        length of the :class:`~.Arc`, but is scaled down to match the start and end
        angles. An exponential function is then applied after it is shifted based on
        the radius of the arc.

    The scaling effect is not applied for arcs with radii smaller than 1 to prevent
    over-scaling.

Parameters
-----
arc
    The :class:`~.Arc` that wraps around the :class:`Brace` mobject.
direction
    The direction from which the brace faces the arc.
    ``LEFT`` for inside the arc, and ``RIGHT`` for the outside.

Example
-----
.. manim:: ArcBraceExample
    :save_last_frame:
    :ref_classes: Arc

    class ArcBraceExample(Scene):
        def construct(self):
            arc_1 = Arc(radius=1.5,start_angle=0,angle=2*PI/3).set_color(RED)
            brace_1 = ArcBrace(arc_1,LEFT)
            group_1 = VGroup(arc_1,brace_1)

            arc_2 = Arc(radius=3,start_angle=0,angle=5*PI/6).set_color(YELLOW)
            brace_2 = ArcBrace(arc_2)
            group_2 = VGroup(arc_2,brace_2)

            arc_3 = Arc(radius=0.5,start_angle=-0,angle=PI).set_color(BLUE)
            brace_3 = ArcBrace(arc_3)
            group_3 = VGroup(arc_3,brace_3)

            arc_4 = Arc(radius=0.2,start_angle=0,angle=3*PI/2).set_color(GREEN)
            brace_4 = ArcBrace(arc_4)
            group_4 = VGroup(arc_4,brace_4)

            arc_group = VGroup(group_1, group_2, group_3, group_4).arrange_in_grid(buff=1.5)
            self.add(arc_group.center())

"""

def __init__(
    self,
    arc: Arc | None = None,
    direction: Sequence[float] = RIGHT,
    **kwargs,
):
    if arc is None:

```

```
arc = Arc(start_angle=-1, angle=2, radius=1)
arc_end_angle = arc.start_angle + arc.angle
line = Line(UP * arc.start_angle, UP * arc_end_angle)
scale_shift = RIGHT * np.log(arc.radius)

if arc.radius >= 1:
    line.scale(arc.radius, about_point=ORIGIN)
    super().__init__(line, direction=direction, **kwargs)
    self.scale(1 / (arc.radius), about_point=ORIGIN)
else:
    super().__init__(line, direction=direction, **kwargs)

if arc.radius >= 0.3:
    self.shift(scale_shift)
else:
    self.shift(RIGHT * np.log(0.3))

self.apply_complex_function(np.exp)
self.shift(arc.get_arc_center())
```

opengl_image_mobject.py

```
from __future__ import annotations

__all__ = [
    "OpenGLImageMobject",
]

from pathlib import Path

import numpy as np
from PIL import Image
from PIL.Image import Resampling

from manim.mobject.opengl.opengl_surface import OpenGLSurface, OpenGLTexturedSurface
from manim.utils.images import get_full_raster_image_path

class OpenGLImageMobject(OpenGLTexturedSurface):
    def __init__(
        self,
        filename_or_array: str | Path | np.ndarray,
        width: float = None,
        height: float = None,
        image_mode: str = "RGBA",
        resampling_algorithm: int = Resampling.BICUBIC,
        opacity: float = 1,
        gloss: float = 0,
        shadow: float = 0,
        **kwargs,
    ):
        self.image = filename_or_array
        self.resampling_algorithm = resampling_algorithm
        if type(filename_or_array) == np.ndarray:
            self.size = self.image.shape[1:-1]
        elif isinstance(filename_or_array, (str, Path)):
            path = get_full_raster_image_path(filename_or_array)
            self.size = Image.open(path).size

        if width is None and height is None:
            width = 4 * self.size[0] / self.size[1]
            height = 4
        if height is None:
            height = width * self.size[1] / self.size[0]
        if width is None:
            width = height * self.size[0] / self.size[1]

        surface = OpenGLSurface(
            lambda u, v: np.array([u, v, 0]),
            [-width / 2, width / 2],
            [-height / 2, height / 2],
            opacity=opacity,
            gloss=gloss,
            shadow=shadow,
        )

        super().__init__(
            surface,
            self.image,
            image_mode=image_mode,
            opacity=opacity,
            gloss=gloss,
            shadow=shadow,
            **kwargs,
        )

    def get_image_from_file(
        self,
        image_file: str | Path | np.ndarray,
        image_mode: str,
    ):
        if isinstance(image_file, (str, Path)):
            return super().get_image_from_file(image_file, image_mode)
        else:
            return (
                Image.fromarray(image_file.astype("uint8"))
                .convert(image_mode)
                .resize(
                    np.array(image_file.shape[:2])
                    * 200, # assumption of 200 ppmu (pixels per manim unit) would suffice
                    resample=self.resampling_algorithm,
                )
            )
```


file_ops.py

```
"""Utility functions for interacting with the file system."""

from __future__ import annotations

__all__ = [
    "add_extension_if_not_present",
    "guarantee_existence",
    "guarantee_empty_existence",
    "seek_full_path_from_defaults",
    "modify_atime",
    "open_file",
    "is_mp4_format",
    "is_gif_format",
    "is_png_format",
    "is_webm_format",
    "is_mov_format",
    "write_to_movie",
    "ensure_executable",
]

import os
import platform
import shutil
import subprocess as sp
import time
from pathlib import Path
from shutil import copyfile
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from ..scene.scene_file_writer import SceneFileWriter

from manim import __version__, config, logger
from .. import console

def is_mp4_format() -> bool:
    """
    Determines if output format is .mp4

    Returns
    ------
    class:`bool`
        ``True`` if format is set as mp4

    """
    return config["format"] == "mp4"

def is_gif_format() -> bool:
    """
    Determines if output format is .gif

    Returns
    ------
    class:`bool`
        ``True`` if format is set as gif

    """
    return config["format"] == "gif"

def is_webm_format() -> bool:
    """
    Determines if output format is .webm

    Returns
    ------
    class:`bool`
        ``True`` if format is set as webm

    """
    return config["format"] == "webm"

def is_mov_format() -> bool:
    """
    Determines if output format is .mov

    Returns
    ------
    class:`bool`
        ``True`` if format is set as mov

    """
    return config["format"] == "mov"

def is_png_format() -> bool:
    """
    Determines if output format is .png

    Returns
    ------
    class:`bool`
        ``True`` if format is set as png

    """
    return config["format"] == "png"

def write_to_movie() -> bool:
```

```

"""
Determines from config if the output is a video format such as mp4 or gif, if the --format is set as 'png'
then it will take precedence even if the write_to_movie flag is set

Returns
-----
class:`bool`
    ``True`` if the output should be written in a movie format

"""
if is_png_format():
    return False
return (
    config["write_to_movie"]
    or is_mp4_format()
    or is_gif_format()
    or is_webm_format()
    or is_mov_format()
)

def ensure_executable(path_to_exe: Path) -> bool:
    if path_to_exe.parent == Path("."):
        executable = shutil.which(path_to_exe.stem)
        if executable is None:
            return False
    else:
        executable = path_to_exe
    return os.access(executable, os.X_OK)

def add_extension_if_not_present(file_name: Path, extension: str) -> Path:
    if file_name.suffix != extension:
        return file_name.with_suffix(file_name.suffix + extension)
    else:
        return file_name

def add_version_before_extension(file_name: Path) -> Path:
    return file_name.with_name(
        f"{file_name.stem}_ManimCE_v{__version__}{file_name.suffix}"
    )

def guarantee_existence(path: Path) -> Path:
    if not path.exists():
        path.mkdir(parents=True)
    return path.resolve(strict=True)

def guarantee_empty_existence(path: Path) -> Path:
    if path.exists():
        shutil.rmtree(str(path))
    path.mkdir(parents=True)
    return path.resolve(strict=True)

def seek_full_path_from_defaults(
    file_name: str, default_dir: Path, extensions: list[str]
) -> Path:
    possible_paths = [Path(file_name).expanduser()]
    possible_paths += [
        Path(default_dir) / f"{file_name}{extension}" for extension in ["", *extensions]
    ]
    for path in possible_paths:
        if path.exists():
            return path
    error = (
        f"From: {Path.cwd()}, could not find {file_name} at either "
        f"of these locations: {list(map(str, possible_paths))}"
    )
    raise OSError(error)

def modify_atime(file_path: str) -> None:
    """Will manually change the accessed time (called `atime`) of the file, as on a lot of OS the accessed time refresh is disabled by default.

    Parameters
    -----
    file_path
        The path of the file.
    """
    os.utime(file_path, times=(time.time(), Path(file_path).stat().st_mtime))

def open_file(file_path, in_browser=False):
    current_os = platform.system()
    if current_os == "Windows":
        os.startfile(file_path if not in_browser else file_path.parent)
    else:
        if current_os == "Linux":
            commands = ["xdg-open"]
            file_path = file_path if not in_browser else file_path.parent
        elif current_os.startswith("CYGWIN"):
            commands = ["cygstart"]
            file_path = file_path if not in_browser else file_path.parent
        elif current_os == "Darwin":
            if is_gif_format():
                commands = ["ffplay", "-loglevel", config["ffmpeg_loglevel"].lower()]
            else:
                commands = ["open"] if not in_browser else ["open", "-R"]
        else:
            raise OSError("Unable to identify your operating system...")
    commands.append(file_path)
    sp.Popen(commands)

def open_media_file(file_writer: SceneFileWriter) -> None:
    file_paths = []

```

```

if config["save_last_frame"]:
    file_paths.append(file_writer.image_file_path)
if write_to_movie() and not is_gif_format():
    file_paths.append(file_writer.movie_file_path)
if write_to_movie() and is_gif_format():
    file_paths.append(file_writer.gif_file_path)

for file_path in file_paths:
    if config["show_in_file_browser"]:
        open_file(file_path, True)
    if config["preview"]:
        open_file(file_path, False)

    logger.info(f"Previewed File at: '{file_path}'")

def get_template_names() -> list[str]:
    """Returns template names from the templates directory.

    Returns
    -------
    :class:`list`
    """
    template_path = Path.resolve(Path(__file__).parent.parent / "templates")
    return [template_name.stem for template_name in template_path.glob("*.mtp")]

def get_template_path() -> Path:
    """Returns the Path of templates directory.

    Returns
    -------
    :class:`Path`
    """
    return Path.resolve(Path(__file__).parent.parent / "templates")

def add_import_statement(file: Path):
    """Prepends an import statement in a file

    Parameters
    -----
    file
    """
    with file.open("r+") as f:
        import_line = "from manim import *"
        content = f.read()
        f.seek(0)
        f.write(import_line + "\n" + content)

def copy_template_files(
    project_dir: Path = Path("."), template_name: str = "Default"
) -> None:
    """Copies template files from templates dir to project_dir.

    Parameters
    -----
    project_dir
        Path to project directory.
    template_name
        Name of template.
    """
    template_cfg_path = Path.resolve(
        Path(__file__).parent.parent / "templates/template.cfg",
    )
    template_scene_path = Path.resolve(
        Path(__file__).parent.parent / f"templates/{template_name}.mtp",
    )

    if not template_cfg_path.exists():
        raise FileNotFoundError(f"{template_cfg_path} : file does not exist")
    if not template_scene_path.exists():
        raise FileNotFoundError(f"{template_scene_path} : file does not exist")

    copyfile(template_cfg_path, Path.resolve(project_dir / "manim.cfg"))
    console.print("\n\t[green]copied[/green] [blue]manim.cfg[/blue]\n")
    copyfile(template_scene_path, Path.resolve(project_dir / "main.py"))
    console.print("\n\t[green]copied[/green] [blue]main.py[/blue]\n")
    add_import_statement(Path.resolve(project_dir / "main.py"))

```

growing.py

```
"""Animations that introduce mobjects to scene by growing them from points.

.. manim:: Growing

    class Growing(Scene):
        def construct(self):
            square = Square()
            circle = Circle()
            triangle = Triangle()
            arrow = Arrow(LEFT, RIGHT)
            star = Star()

            VGroup(square, circle, triangle).set_x(0).arrange(buff=1.5).set_y(2)
            VGroup(arrow, star).move_to(DOWN).set_x(0).arrange(buff=1.5).set_y(-2)

            self.play(GrowFromPoint(square, ORIGIN))
            self.play(GrowFromCenter(circle))
            self.play(GrowFromEdge(triangle, DOWN))
            self.play(GrowArrow(arrow))
            self.play(SpinInFromNothing(star))

"""

from __future__ import annotations

__all__ = [
    "GrowFromPoint",
    "GrowFromCenter",
    "GrowFromEdge",
    "GrowArrow",
    "SpinInFromNothing",
]
import typing
import numpy as np

from ..animation.transform import Transform
from ..constants import PI
from ..utils.paths import spiral_path

if typing.TYPE_CHECKING:
    from manim.mobject.geometry.line import Arrow

    from ..mobject.mobject import Mobject

class GrowFromPoint(Transform):
    """Introduce an :class:`~.Mobject` by growing it from a point.

    Parameters
    -----
    mobject
        The mobjects to be introduced.
    point
        The point from which the mobject grows.
    point_color
        Initial color of the mobject before growing to its full size. Leave empty to match mobject's color.

    Examples
    -----
    .. manim :: GrowFromPointExample

        class GrowFromPointExample(Scene):
            def construct(self):
                dot = Dot(3 * UR, color=GREEN)
                squares = [Square() for _ in range(4)]
                VGroup(*squares).set_x(0).arrange(buff=1)
                self.add(dot)
                self.play(GrowFromPoint(squares[0], ORIGIN))
                self.play(GrowFromPoint(squares[1], [-2, 2, 0]))
                self.play(GrowFromPoint(squares[2], [3, -2, 0], RED))
                self.play(GrowFromPoint(squares[3], dot, dot.get_color()))

    """

    def __init__(self, mobject: Mobject, point: np.ndarray, point_color: str = None, **kwargs
) -> None:
        self.point = point
```

```

        self.point_color = point_color
    super().__init__(mobject, introducer=True, **kwargs)

def create_target(self) -> Mobject:
    return self.mobject

def create_starting_mobject(self) -> Mobject:
    start = super().create_starting_mobject()
    start.scale(0)
    start.move_to(self.point)
    if self.point_color:
        start.set_color(self.point_color)
    return start


class GrowFromCenter(GrowFromPoint):
    """Introduce an :class:`~.Mobject` by growing it from its center.

    Parameters
    -----
    mobject
        The mobjects to be introduced.
    point_color
        Initial color of the mobject before growing to its full size. Leave empty to match mobject's color.

    Examples
    -----
    .. manim :: GrowFromCenterExample

    class GrowFromCenterExample(Scene):
        def construct(self):
            squares = [Square() for _ in range(2)]
            VGroup(*squares).set_x(0).arrange(buff=2)
            self.play(GrowFromCenter(squares[0]))
            self.play(GrowFromCenter(squares[1], point_color=RED))

    """

    def __init__(self, mobject: Mobject, point_color: str = None, **kwargs) -> None:
        point = mobject.get_center()
        super().__init__(mobject, point, point_color=point_color, **kwargs)


class GrowFromEdge(GrowFromPoint):
    """Introduce an :class:`~.Mobject` by growing it from one of its bounding box edges.

    Parameters
    -----
    mobject
        The mobjects to be introduced.
    edge
        The direction to seek bounding box edge of mobject.
    point_color
        Initial color of the mobject before growing to its full size. Leave empty to match mobject's color.

    Examples
    -----
    .. manim :: GrowFromEdgeExample

    class GrowFromEdgeExample(Scene):
        def construct(self):
            squares = [Square() for _ in range(4)]
            VGroup(*squares).set_x(0).arrange(buff=1)
            self.play(GrowFromEdge(squares[0], DOWN))
            self.play(GrowFromEdge(squares[1], RIGHT))
            self.play(GrowFromEdge(squares[2], UR))
            self.play(GrowFromEdge(squares[3], UP, point_color=RED))

    """

    def __init__(
        self, mobject: Mobject, edge: np.ndarray, point_color: str = None, **kwargs
    ) -> None:
        point = mobject.get_critical_point(edge)
        super().__init__(mobject, point, point_color=point_color, **kwargs)


class GrowArrow(GrowFromPoint):
    """Introduce an :class:`~.Arrow` by growing it from its start toward its tip.

    Parameters
    -----
    arrow
        The arrow to be introduced.

```

```
point_color
    Initial color of the arrow before growing to its full size. Leave empty to match arrow's color.
```

Examples

```
.. manim :: GrowArrowExample
```

```
class GrowArrowExample(Scene):
    def construct(self):
        arrows = [Arrow(2 * LEFT, 2 * RIGHT), Arrow(2 * DR, 2 * UL)]
        VGroup(*arrows).set_x(0).arrange(buff=2)
        self.play(GrowArrow(arrows[0]))
        self.play(GrowArrow(arrows[1], point_color=RED))
```

"""

```
def __init__(self, arrow: Arrow, point_color: str = None, **kwargs) -> None:
    point = arrow.get_start()
    super().__init__(arrow, point, point_color=point_color, **kwargs)
```

```
def create_starting_mobject(self) -> Mobject:
    start_arrow = self.mobject.copy()
    start_arrow.scale(0, scale_tips=True, about_point=self.point)
    if self.point_color:
        start_arrow.set_color(self.point_color)
    return start_arrow
```

```
class SpinInFromNothing(GrowFromCenter):
```

"""Introduce an :class:`~.Mobject` spinning and growing it from its center.

Parameters

mobject

The mobjects to be introduced.

angle

The amount of spinning before mobject reaches its full size. E.g. 2π means that the object will do one full spin before being fully introduced.

point_color

Initial color of the mobject before growing to its full size. Leave empty to match mobject's color.

Examples

```
.. manim :: SpinInFromNothingExample
```

```
class SpinInFromNothingExample(Scene):
    def construct(self):
        squares = [Square() for _ in range(3)]
        VGroup(*squares).set_x(0).arrange(buff=2)
        self.play(SpinInFromNothing(squares[0]))
        self.play(SpinInFromNothing(squares[1], angle=2 * PI))
        self.play(SpinInFromNothing(squares[2], point_color=RED))
```

"""

```
def __init__(
    self, mobject: Mobject, angle: float = PI / 2, point_color: str = None, **kwargs
) -> None:
    self.angle = angle
    super().__init__(
        mobject, path_func=spiral_path(angle), point_color=point_color, **kwargs
    )
```

table.py

```
r"""Mobjects representing tables.

Examples
-----
.. manim:: TableExamples
    :save_last_frame:

class TableExamples(Scene):
    def construct(self):
        t0 = Table(
            [["First", "Second"],
             ["Third", "Fourth"]],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")],
            top_left_entry=Text("TOP"))
        t0.add_highlighted_cell((2,2), color=GREEN)
        x_vals = np.linspace(-2,2,5)
        y_vals = np.exp(x_vals)
        t1 = DecimalTable(
            [x_vals, y_vals],
            row_labels=[MathTex("x"), MathTex("f(x)"),
            include_outer_lines=True])
        t1.add(t1.get_cell((2,2), color=RED))
        t2 = MathTable(
            [{"": 0, 5, 10},
             [0, 0, 5, 10],
             [2, 2, 7, 12],
             [4, 4, 9, 14],
             include_outer_lines=True])
        t2.get_horizontal_lines()[3].set_color(BLUE)
        t2.get_vertical_lines()[3].set_color(BLUE)
        t2.get_horizontal_lines()[3].set_z_index(1)
        cross = VGroup(
            Line(UP + LEFT, DOWN + RIGHT),
            Line(UP + RIGHT, DOWN + LEFT))
        a = Circle().set_color(RED).scale(0.5)
        b = cross.set_color(BLUE).scale(0.5)
        t3 = MobjectTable(
            [[a.copy(),b.copy(),a.copy()],
             [b.copy(),a.copy(),a.copy()],
             [a.copy(),b.copy(),b.copy()]])
        t3.add(Line(
            t3.get_corner(DL), t3.get_corner(UR),
            .set_color(RED)))
        vals = np.arange(1,21).reshape(5,4)
        t4 = IntegerTable(
            vals,
            include_outer_lines=True)
        g1 = Group(t0, t1).scale(0.5).arrange(buff=1).to_edge(UP, buff=1)
        g2 = Group(t2, t3, t4).scale(0.5).arrange(buff=1).to_edge(DOWN, buff=1)
        self.add(g1, g2)
    """

from __future__ import annotations

__all__ = [
    "Table",
    "MathTable",
    "MobjectTable",
    "IntegerTable",
    "DecimalTable",
]

import itertools as it
from typing import Callable, Iterable, Sequence

from colour import Color

from manim.mobject.geometry.line import Line
from manim.mobject.geometry.polygram import Polygon
from manim.mobject.geometry.shape_matchers import BackgroundRectangle
from manim.mobject.text.numbers import DecimalNumber, Integer
from manim.mobject.tex.tex_mobject import MathTex
from manim.mobject.text.text_mobject import Paragraph

from .. import config
from ..animation.animation import Animation
from ..animation.composition import AnimationGroup
from ..animation.creation import Create, Write
from ..animation.fading import FadeIn
from ..mobject.types.vectorized_mobject import VGroup, VMobject
from ..utils.color import BLACK, YELLOW
from .utils import get_vectorized_mobject_class

class Table(VGroup):
    """A mobject that displays a table on the screen.

    Parameters
    -----
    table
        A 2D array or list of lists. Content of the table has to be a valid input
        for the callable set in ``element_to_mobject``.
    row_labels
        List of :class:`~.VMobject` representing the labels of each row.
    col_labels
        List of :class:`~.VMobject` representing the labels of each column.
    top_left_entry
        The top-left entry of the table, can only be specified if row and
        column labels are given.
    v_buff
        Vertical buffer passed to :meth:`~.Mobject.arrange_in_grid`, by default 0.8.
    h_buff
        Horizontal buffer passed to :meth:`~.Mobject.arrange_in_grid`, by default 1.3.
    include_outer_lines
        ``True`` if the table should include outer lines, by default False.
    add_background_rectangles_to_entries
        ``True`` if background rectangles should be added to entries, by default ``False``.
    entries_background_color
        Background color of entries if ``add_background_rectangles_to_entries`` is ``True``.
    include_background_rectangle
        ``True`` if the table should have a background rectangle, by default ``False``.
    background_rectangle_color
        Background color of table if ``include_background_rectangle`` is ``True``.
    element_to_mobject
        The :class:`~.Mobject` class applied to the table entries. by default :class:`~.Paragraph`. For common choices, see :mod:`~.text_mobject`/:mod:`~.tex_mobject`.
    element_to_mobject_config
        Custom configuration passed to :attr:`element_to_mobject`, by default {}.
    
```

```

arrange_in_grid_config
    Dict passed to :meth:`~.Mobject.arrange_in_grid`, customizes the arrangement of the table.
line_config
    Dict passed to :class:`~.Line`, customizes the lines of the table.
kwargs
    Additional arguments to be passed to :class:`~.VGroup`.

Examples
-----
.. manim:: TableExamples
    :save_last_frame:

    class TableExamples(Scene):
        def construct(self):
            t0 = Table(
                [{"This", "is a"}, {"simple", "Table in \n Manim."}])
            t1 = Table(
                [{"This", "is a"}, {"simple", "Table."}],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")])
            t1.add_highlighted_cell((2,2), color=YELLOW)
            t2 = Table(
                [{"This", "is a"}, {"simple", "Table."}],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")],
                top_left_entry=Star().scale(0.3),
                include_outer_lines=True,
                arrange_in_grid_config={"cell_alignment": RIGHT})
            t2.add(t2.get_cell((2,2), color=RED))
            t3 = Table(
                [{"This", "is a"}, {"simple", "Table."}],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")],
                top_left_entry=Star().scale(0.3),
                include_outer_lines=True,
                line_config={"stroke_width": 1, "color": YELLOW})
            t3.remove(*t3.get_vertical_lines())
            g = Group(
                t0,t1,t2,t3
            ).scale(0.7).arrange_in_grid(buff=1)
            self.add(g)

.. manim:: BackgroundRectanglesExample
    :save_last_frame:

    class BackgroundRectanglesExample(Scene):
        def construct(self):
            background = Rectangle(height=6.5, width=13)
            background.set_fill(opacity=.5)
            background.set_color([TEAL, RED, YELLOW])
            self.add(background)
            t0 = Table(
                [{"This", "is a"}, {"simple", "Table."}],
                add_background_rectangles_to_entries=True)
            t1 = Table(
                [{"This", "is a"}, {"simple", "Table."}],
                include_background_rectangle=True)
            g = Group(t0, t1).scale(0.7).arrange(buff=0.5)
            self.add(g)
    """

def __init__(
    self,
    table: Iterable[Iterable[float | str | VMobject]],
    row_labels: Iterable[VMobject] | None = None,
    col_labels: Iterable[VMobject] | None = None,
    top_left_entry: VMobject | None = None,
    v_buff: float = 0.8,
    h_buff: float = 1.3,
    include_outer_lines: bool = False,
    add_background_rectangles_to_entries: bool = False,
    entries_background_color: Color = BLACK,
    include_background_rectangle: bool = False,
    background_rectangle_color: Color = BLACK,
    element_to_mobject: Callable[
        [float | str | VMobject],
        VMobject,
    ] = Paragraph,
    element_to_mobject_config: dict = {},
    arrange_in_grid_config: dict = {},
    line_config: dict = {},
    **kwargs,
):
    self.row_labels = row_labels
    self.col_labels = col_labels
    self.top_left_entry = top_left_entry
    self.row_dim = len(table)
    self.col_dim = len(table[0])
    self.v_buff = v_buff
    self.h_buff = h_buff
    self.include_outer_lines = include_outer_lines
    self.add_background_rectangles_to_entries = add_background_rectangles_to_entries
    self.entries_background_color = entries_background_color
    self.include_background_rectangle = include_background_rectangle
    self.background_rectangle_color = background_rectangle_color
    self.element_to_mobject = element_to_mobject
    self.element_to_mobject_config = element_to_mobject_config
    self.arrange_in_grid_config = arrange_in_grid_config
    self.line_config = line_config

    for row in table:
        if len(row) == len(table[0]):
            pass
        else:
            raise ValueError("Not all rows in table have the same length.")

    super().__init__(**kwargs)
    mob_table = self._table_to_mob_table(table)
    self.elements_without_labels = VGroup(*it.chain(*mob_table))
    mob_table = self._add_labels(mob_table)
    self._organize_mob_table(mob_table)
    self.elements = VGroup(*it.chain(*mob_table))

    if len(self.elements[0].get_all_points()) == 0:
        self.elements.remove(self.elements[0])

    self.add(self.elements)
    self.center()
    self.mob_table = mob_table
    self._add_horizontal_lines()

```

```

self._add_vertical_lines()
if self._add_background_rectangles_to_entries:
    self._add_background_to_entries(color=self.entries_background_color)
if self._include_background_rectangle:
    self._add_background_rectangle(color=self.background_rectangle_color)

def _table_to_mob_table(
    self,
    table: Iterable[Iterable[float | str | VMobject]],
) -> list:
    """Initializes the entries of ``table`` as :class:`~.VMobject`.

    Parameters
    -----
    table
        A 2D array or list of lists. Content of the table has to be a valid input
        for the callable set in ``element_to_mobobject``.

    Returns
    -----
    List
        List of :class:`~.VMobject` from the entries of ``table``.
    """
    return [
        self.element_to_mobobject(item, **self.element_to_mobobject_config)
        for item in row
    ]
    for row in table
]

def _organize_mob_table(self, table: Iterable[Iterable[VMobject]]) -> VGroup:
    """Arranges the :class:`~.VMobject` of ``table`` in a grid.

    Parameters
    -----
    table
        A 2D iterable object with :class:`~.VMobject` entries.

    Returns
    -----
    :class:`~.VGroup`
        The :class:`~.VMobject` of the ``table`` in a :class:`~.VGroup` already
        arranged in a table-like grid.
    """
    help_table = VGroup()
    for i, row in enumerate(table):
        for j, _ in enumerate(row):
            help_table.add(table[i][j])
    help_table.arrange_in_grid(
        rows=len(table),
        cols=len(table[0]),
        buff=(self.h_buff, self.v_buff),
        **self.arrange_in_grid_config,
    )
    return help_table

def _add_labels(self, mob_table: VGroup) -> VGroup:
    """Adds labels to an in a grid arranged :class:`~.VGroup`.

    Parameters
    -----
    mob_table
        An in a grid organized class:`~.VGroup`.

    Returns
    -----
    :class:`~.VGroup`
        Returns the ``mob_table`` with added labels.
    """
    if self.row_labels is not None:
        for k in range(len(self.row_labels)):
            mob_table[k] = [self.row_labels[k]] + mob_table[k]
    if self.col_labels is not None:
        if self.row_labels is not None:
            if self.top_left_entry is not None:
                col_labels = [self.top_left_entry] + self.col_labels
                mob_table.insert(0, col_labels)
            else:
                # Placeholder to use arrange_in_grid if top_left_entry is not set.
                # Import OpenGLOMObject to work with --renderer=openGL
                dummy_mobject = get_vectorized_mobject_class()()
                col_labels = [dummy_mobject] + self.col_labels
                mob_table.insert(0, col_labels)
        else:
            mob_table.insert(0, self.col_labels)
    return mob_table

def _add_horizontal_lines(self) -> Table:
    """Adds the horizontal lines to the table."""
    anchor_left = self.get_left()[0] - 0.5 * self.h_buff
    anchor_right = self.get_right()[0] + 0.5 * self.h_buff
    line_group = VGroup()
    if self._include_outer_lines:
        anchor = self.get_rows()[0].get_top()[1] + 0.5 * self.v_buff
        line = Line(
            [anchor_left, anchor, 0], [anchor_right, anchor, 0], **self.line_config
        )
        line_group.add(line)
        self.add(line)
        anchor = self.get_rows()[-1].get_bottom()[1] - 0.5 * self.v_buff
        line = Line(
            [anchor_left, anchor, 0], [anchor_right, anchor, 0], **self.line_config
        )
        line_group.add(line)
        self.add(line)
    for k in range(len(self.mob_table) - 1):
        anchor = self.get_rows()[k + 1].get_top()[1] + 0.5 * (
            self.get_rows()[k].get_bottom()[1] - self.get_rows()[k + 1].get_top()[1]
        )
        line = Line(
            [anchor_left, anchor, 0], [anchor_right, anchor, 0], **self.line_config
        )
        line_group.add(line)
        self.add(line)
    self.horizontal_lines = line_group
    return self

def _add_vertical_lines(self) -> Table:
    """Adds the vertical lines to the table"""
    anchor_top = self.get_rows().get_top()[1] + 0.5 * self.v_buff
    anchor_bottom = self.get_rows().get_bottom()[1] - 0.5 * self.v_buff
    line_group = VGroup()
    if self._include_outer_lines:
        anchor = self.get_columns()[0].get_left()[0] - 0.5 * self.h_buff
        line = Line(
            [anchor, anchor_top, 0], [anchor, anchor_bottom, 0], **self.line_config
        )

```

```

        )
    line_group.add(line)
    self.add(line)
    anchor = self.get_columns()[-1].get_right()[0] + 0.5 * self.h_buff
    line = Line(
        [anchor, anchor_top, 0], [anchor, anchor_bottom, 0], **self.line_config
    )
    line_group.add(line)
    self.add(line)
for k in range(len(self.mob_table[0]) - 1):
    anchor = self.get_columns()[k + 1].get_left()[0] + 0.5 * (
        self.get_columns()[k].get_right()[0]
        - self.get_columns()[k + 1].get_left()[0]
    )
    line = Line(
        [anchor, anchor_bottom, 0], [anchor, anchor_top, 0], **self.line_config
    )
    line_group.add(line)
    self.add(line)
self.vertical_lines = line_group
return self

def get_horizontal_lines(self) -> VGroup:
    """Return the horizontal lines of the table.

    Returns
    ------
    :class:`~.VGroup`
        :class:`~.VGroup` containing all the horizontal lines of the table.

    Examples
    -----
    .. manim:: GetHorizontalLinesExample
        :save_last_frame:

    class GetHorizontalLinesExample(Scene):
        def construct(self):
            table = Table(
                [["First", "Second"],
                 ["Third", "Fourth"]],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")])
            table.get_horizontal_lines().set_color(RED)
            self.add(table)
    """
    return self.horizontal_lines

def get_vertical_lines(self) -> VGroup:
    """Return the vertical lines of the table.

    Returns
    ------
    :class:`~.VGroup`
        :class:`~.VGroup` containing all the vertical lines of the table.

    Examples
    -----
    .. manim:: GetVerticalLinesExample
        :save_last_frame:

    class GetVerticalLinesExample(Scene):
        def construct(self):
            table = Table(
                [["First", "Second"],
                 ["Third", "Fourth"]],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")])
            table.get_vertical_lines()[0].set_color(RED)
            self.add(table)
    """
    return self.vertical_lines

def get_columns(self) -> VGroup:
    """Return columns of the table as a :class:`~.VGroup` of :class:`~.VGroup`.

    Returns
    ------
    :class:`~.VGroup`
        :class:`~.VGroup` containing each column in a :class:`~.VGroup`.

    Examples
    -----
    .. manim:: GetColumnsExample
        :save_last_frame:

    class GetColumnsExample(Scene):
        def construct(self):
            table = Table(
                [["First", "Second"],
                 ["Third", "Fourth"]],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")])
            table.add(SurroundingRectangle(table.get_columns()[1]))
            self.add(table)
    """
    return VGroup(
        *((VGroup(*[row[i] for row in self.mob_table])
            for i in range(len(self.mob_table[0]))))
    )

def get_rows(self) -> VGroup:
    """Return the rows of the table as a :class:`~.VGroup` of :class:`~.VGroup`.

    Returns
    ------
    :class:`~.VGroup`
        :class:`~.VGroup` containing each row in a :class:`~.VGroup`.

    Examples
    -----
    .. manim:: GetRowsExample
        :save_last_frame:

    class GetRowsExample(Scene):
        def construct(self):
            table = Table(
                [["First", "Second"],
                 ["Third", "Fourth"]],
                row_labels=[Text("R1"), Text("R2")],
                col_labels=[Text("C1"), Text("C2")])
            table.add(SurroundingRectangle(table.get_rows()[1]))

```

```

        self.add(table)
"""
return VGroup(*(VGroup(*row) for row in self.mob_table))

def set_column_colors(self, *colors: Iterable[Color]) -> Table:
    """Set individual colors for each column of the table.

Parameters
-----
colors
    An iterable of colors; each color corresponds to a column.

Examples
-----
.. manim:: SetColumnColorsExample
:save_last_frame:

class SetColumnColorsExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        ).set_column_colors([RED,BLUE], GREEN)
        self.add(table)

    """
columns = self.get_columns()
for color, column in zip(colors, columns):
    column.set_color(color)
return self

def set_row_colors(self, *colors: Iterable[Color]) -> Table:
    """Set individual colors for each row of the table.

Parameters
-----
colors
    An iterable of colors; each color corresponds to a row.

Examples
-----
.. manim:: SetRowColorsExample
:save_last_frame:

class SetRowColorsExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        ).set_row_colors([RED,BLUE], GREEN)
        self.add(table)

    """
rows = self.get_rows()
for color, row in zip(colors, rows):
    row.set_color(color)
return self

def get_entries(
    self,
    pos: Sequence[int] | None = None,
) -> VMobject | VGroup:
    """Return the individual entries of the table (including labels) or one specific entry
    if the parameter, ``pos`` , is set.

Parameters
-----
pos
    The position of a specific entry on the table. ``(1,1)`` being the top left entry
    of the table.

Returns
-----
Union[:class:`~.VMobject` , :class:`~.VGroup`]
    :class:`~.VGroup` containing all entries of the table (including labels)
    or the :class:`~.VMobject` at the given position if ``pos`` is set.

Examples
-----
.. manim:: GetEntriesExample
:save_last_frame:

class GetEntriesExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        ent = table.get_entries()
        for item in ent:
            item.set_color(random_bright_color())
        table.get_entries((2,2)).rotate(PI)
        self.add(table)

    """
if pos is not None:
    if (
        self.row_labels is not None
        and self.col_labels is not None
        and self.top_left_entry is None
    ):
        index = len(self.mob_table[0]) * (pos[0] - 1) + pos[1] - 2
        return self.elements[index]
    else:
        index = len(self.mob_table[0]) * (pos[0] - 1) + pos[1] - 1
        return self.elements[index]
else:
    return self.elements

def get_entries_without_labels(
    self,
    pos: Sequence[int] | None = None,
) -> VMobject | VGroup:
    """Return the individual entries of the table (without labels) or one specific entry
    if the parameter, ``pos`` , is set.

Parameters
-----
pos
    The position of a specific entry on the table. ``(1,1)`` being the top left entry
    of the table (without labels).

```

```

>Returns
-----
Union[class: `~.VMobject`, class: `~.VGroup`]
:class:`~.VGroup` containing all entries of the table (without labels)
or the :class:`~.VMobject` at the given position if ``pos`` is set.

Examples
-----
.. manim:: GetEntriesWithoutLabelsExample
:save_last_frame:

class GetEntriesWithoutLabelsExample(Scene):
    def construct(self):
        table = Table(
            [{"first", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        ent = table.get_entries_without_labels()
        colors = [BLUE, GREEN, YELLOW, RED]
        for k in range(len(colors)):
            ent[k].set_color(colors[k])
        table.get_entries_without_labels((2,2)).rotate(PI)
        self.add(table)

    """
    if pos is not None:
        index = self.col_dim * (pos[0] - 1) + pos[1] - 1
        return self.elements_without_labels[index]
    else:
        return self.elements_without_labels

def get_row_labels(self) -> VGroup:
    """Return the row labels of the table.

>Returns
-----
:class:`~.VGroup`
:class:`~.VGroup` containing the row labels of the table.

Examples
-----
.. manim:: GetRowLabelsExample
:save_last_frame:

class GetRowLabelsExample(Scene):
    def construct(self):
        table = Table(
            [{"first", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        lab = table.get_row_labels()
        for item in lab:
            item.set_color(random_bright_color())
        self.add(table)

    """
    return VGroup(*self.row_labels)

def get_col_labels(self) -> VGroup:
    """Return the column labels of the table.

>Returns
-----
:class:`~.VGroup`
VGroup containing the column labels of the table.

Examples
-----
.. manim:: GetColLabelsExample
:save_last_frame:

class GetColLabelsExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        lab = table.get_col_labels()
        for item in lab:
            item.set_color(random_bright_color())
        self.add(table)

    """
    return VGroup(*self.col_labels)

def get_labels(self) -> VGroup:
    """Returns the labels of the table.

>Returns
-----
:class:`~.VGroup`
:class:`~.VGroup` containing all the labels of the table.

Examples
-----
.. manim:: GetLabelsExample
:save_last_frame:

class GetLabelsExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        lab = table.get_labels()
        colors = [BLUE, GREEN, YELLOW, RED]
        for k in range(len(colors)):
            lab[k].set_color(colors[k])
        self.add(table)

    """
    label_group = VGroup()
    if self.top_left_entry is not None:
        label_group.add(self.top_left_entry)
    for label in (self.col_labels, self.row_labels):
        if label is not None:
            label_group.add(*label)
    return label_group

def add_background_to_entries(self, color: Color = BLACK) -> Table:
    """Adds a black :class:`~.BackgroundRectangle` to each entry of the table."""

```

```

for mob in self.get_entries():
    mob.add_background_rectangle(color=color)
return self

def get_cell(self, pos: Sequence[int] = (1, 1), **kwargs) -> Polygon:
    """Returns one specific cell as a rectangular :class:`~.Polygon` without the entry.

Parameters
-----
pos
    The position of a specific entry on the table. ``(1,1)`` being the top left entry
    of the table.
kwargs
    Additional arguments to be passed to :class:`~.Polygon`.

Returns
-----
:class:`~.Polygon`
    Polygon mimicking one specific cell of the Table.

Examples
-----
.. manim:: GetCellExample
:save_last_frame:

class GetCellExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        cell = table.get_cell((2,2), color=RED)
        self.add(table, cell)

    row = self.get_rows()[pos[0] - 1]
    col = self.get_columns()[pos[1] - 1]
    edge_UL = [
        col.get_left()[0] - self.h_buff / 2,
        row.get_top()[1] + self.v_buff / 2,
        0,
    ]
    edge_UR = [
        col.get_right()[0] + self.h_buff / 2,
        row.get_top()[1] + self.v_buff / 2,
        0,
    ]
    edge_DL = [
        col.get_left()[0] - self.h_buff / 2,
        row.get_bottom()[1] - self.v_buff / 2,
        0,
    ]
    edge_DR = [
        col.get_right()[0] + self.h_buff / 2,
        row.get_bottom()[1] - self.v_buff / 2,
        0,
    ]
    rec = Polygon(edge_UL, edge_UR, edge_DR, edge_DL, **kwargs)
    return rec

def get_highlighted_cell(
    self, pos: Sequence[int] = (1, 1), color: Color = YELLOW, **kwargs
) -> BackgroundRectangle:
    """Returns a :class:`~.BackgroundRectangle` of the cell at the given position.

Parameters
-----
pos
    The position of a specific entry on the table. ``(1,1)`` being the top left entry
    of the table.
color
    The color used to highlight the cell.
kwargs
    Additional arguments to be passed to :class:`~.BackgroundRectangle`.

Examples
-----
.. manim:: GetHighlightedCellExample
:save_last_frame:

class GetHighlightedCellExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        highlight = table.get_highlighted_cell((2,2), color=GREEN)
        table.add_to_back(highlight)
        self.add(table)

    cell = self.get_cell(pos)
    bg_cell = BackgroundRectangle(cell, color=color, **kwargs)
    return bg_cell

def add_highlighted_cell(
    self, pos: Sequence[int] = (1, 1), color: Color = YELLOW, **kwargs
) -> Table:
    """Highlights one cell at a specific position on the table by adding a :class:`~.BackgroundRectangle`.

Parameters
-----
pos
    The position of a specific entry on the table. ``(1,1)`` being the top left entry
    of the table.
color
    The color used to highlight the cell.
kwargs
    Additional arguments to be passed to :class:`~.BackgroundRectangle`.

Examples
-----
.. manim:: AddHighlightedCellExample
:save_last_frame:

class AddHighlightedCellExample(Scene):
    def construct(self):
        table = Table(
            [{"First", "Second"}, {"Third", "Fourth"}],
            row_labels=[Text("R1"), Text("R2")],
            col_labels=[Text("C1"), Text("C2")])
        table.add_highlighted_cell((2,2), color=GREEN)
        self.add(table)

```

```

"""
bg_cell = self.get_highlighted_cell(pos, color=color, **kwargs)
self.add_to_back(bg_cell)
entry = self.get_entries(pos)
entry.background_rectangle = bg_cell
return self

def create(
    self,
    lag_ratio: float = 1,
    line_animation: Callable[[VMobject | VGroup], Animation] = Create,
    label_animation: Callable[[VMobject | VGroup], Animation] = Write,
    element_animation: Callable[[VMobject | VGroup], Animation] = Create,
    entry_animation: Callable[[VMobject | VGroup], Animation] = FadeIn,
    **kwargs,
) -> AnimationGroup:
    """Customized create-type function for tables.

    Parameters
    -----
    lag_ratio
        The lag ratio of the animation.
    line_animation
        The animation style of the table lines, see :mod:`~.creation` for examples.
    label_animation
        The animation style of the table labels, see :mod:`~.creation` for examples.
    element_animation
        The animation style of the table elements, see :mod:`~.creation` for examples.
    entry_animation
        The entry animation of the table background, see :mod:`~.creation` for examples.
    kwargs
        Further arguments passed to the creation animations.

    Returns
    -----
    :class:`~.AnimationGroup`
        AnimationGroup containing creation of the lines and of the elements.

    Examples
    -----
    .. manim:: CreateTableExample

        class CreateTableExample(Scene):
            def construct(self):
                table = Table(
                    [{"First", "Second"}, {"Third", "Fourth"}],
                    row_labels=[Text("R1"), Text("R2")],
                    col_labels=[Text("C1"), Text("C2")],
                    include_outer_lines=True)
                self.play(table.create())
                self.wait()

        """
        animations: Sequence[Animation] = [
            line_animation(
                VGroup(self.vertical_lines, self.horizontal_lines),
                **kwargs,
            ),
            element_animation(self.elements_without_labels.set_z_index(2), **kwargs),
        ]

        if self.get_labels():
            animations += [
                label_animation(self.get_labels(), **kwargs),
            ]

        if self.get_entries():
            for entry in self.elements_without_labels:
                try:
                    animations += [
                        entry_animation(
                            entry.background_rectangle,
                            **kwargs,
                        )
                    ]
                except AttributeError:
                    continue

        return AnimationGroup(*animations, lag_ratio=lag_ratio)

    def scale(self, scale_factor: float, **kwargs):
        # h_buff and v_buff must be adjusted so that Table.get_cell
        # can construct an accurate polygon for a cell.
        self.h_buff *= scale_factor
        self.v_buff *= scale_factor
        super().scale(scale_factor, **kwargs)
        return self

class MathTable(Table):
    """A specialized :class:`~.Table` mobject for use with LaTeX.

    Examples
    -----
    .. manim:: MathTableExample
        :save_last_frame:

        class MathTableExample(Scene):
            def construct(self):
                t0 = MathTable(
                    [{"+", 0, 5, 10}, {0, 0, 5, 10}, {2, 2, 7, 12}, {4, 4, 9, 14}],
                    include_outer_lines=True)
                self.add(t0)

        """
        def __init__(self, table: Iterable[Iterable[float | str]], element_to_mobject: Callable[[float | str], VMobject] = MathTex, **kwargs,):
            """
            Special case of :class:`~.Table` with `element_to_mobject` set to :class:`~.MathTex`.
            Every entry in `table` is set in a Latex `align` environment.

            Parameters
            -----
            table
                A 2d array or list of lists. Content of the table have to be valid input
                for :class:`~.MathTex`.
            element_to_mobject
                The :class:`~.Mobject` class applied to the table entries. Set as :class:`~.MathTex`.
            """

```

```

kwargs
    Additional arguments to be passed to :class:`~.Table`.
"""
super().__init__(
    table,
    element_to_mobject=element_to_mobject,
    **kwargs,
)

class MobjectTable(Table):
    """A specialized :class:`~.Table` mobject for use with :class:`~.Mobject`."""

Examples
-----
.. manim:: MobjectTableExample
    :save_last_frame:

class MobjectTableExample(Scene):
    def construct(self):
        cross = VGroup(
            Line(UP + LEFT, DOWN + RIGHT),
            Line(UP + RIGHT, DOWN + LEFT),
        )
        a = Circle().set_color(RED).scale(0.5)
        b = cross.set_color(BLUE).scale(0.5)
        t0 = MobjectTable(
            [[a.copy(),b.copy(),a.copy()],
             [b.copy(),a.copy(),a.copy()],
             [a.copy(),b.copy(),b.copy()]]
        )
        line = Line(
            t0.get_corner(DL), t0.get_corner(UR)
        ).set_color(RED)
        self.add(t0, line)

    def __init__(self,
                 table: Iterable[Iterable[VMobject]],
                 element_to_mobject: Callable[[VMobject], VMobject] = lambda m: m,
                 **kwargs,
                 ):
        """
        Special case of :class:`~.Table` with ``element_to_mobject`` set to an identity function.
        Here, every item in ``table`` must already be of type :class:`~.Mobject`.

        Parameters
        -----
        table
            A 2D array or list of lists. Content of the table must be of type :class:`~.Mobject`.
        element_to_mobject
            The :class:`~.Mobject` class applied to the table entries. Set as ``lambda m : m`` to return itself.
        kwargs
            Additional arguments to be passed to :class:`~.Table`.
        """
        super().__init__(table, element_to_mobject=element_to_mobject, **kwargs)

class IntegerTable(Table):
    """A specialized :class:`~.Table` mobject for use with :class:`~.Integer`."""

Examples
-----
.. manim:: IntegerTableExample
    :save_last_frame:

class IntegerTableExample(Scene):
    def construct(self):
        t0 = IntegerTable(
            [[0,30,45,60,90],
             [90,60,45,30,0]],
            col_labels=[MathTex("\frac{\sqrt{0}}{2}"),
                        MathTex("\frac{\sqrt{1}}{2}"),
                        MathTex("\frac{\sqrt{2}}{2}"),
                        MathTex("\frac{\sqrt{3}}{2}"),
                        MathTex("\frac{\sqrt{4}}{2}"),
            ],
            row_labels=[MathTex("\sin"), MathTex("\cos")],
            h_buff=1,
            element_to_mobject_config={"unit": "\circ"})
        self.add(t0)

    def __init__(self,
                 table: Iterable[Iterable[float | str]],
                 element_to_mobject: Callable[[float | str], VMobject] = Integer,
                 **kwargs,
                 ):
        """
        Special case of :class:`~.Table` with `element_to_mobject` set to :class:`~.Integer`.
        Will round if there are decimal entries in the table.

        Parameters
        -----
        table
            A 2d array or list of lists. Content of the table has to be valid input
            for :class:`~.Integer`.
        element_to_mobject
            The :class:`~.Mobject` class applied to the table entries. Set as :class:`~.Integer`.
        kwargs
            Additional arguments to be passed to :class:`~.Table`.
        """
        super().__init__(table, element_to_mobject=element_to_mobject, **kwargs)

class DecimalTable(Table):
    """A specialized :class:`~.Table` mobject for use with :class:`~.DecimalNumber` to display decimal entries."""

Examples
-----
.. manim:: DecimalTableExample
    :save_last_frame:

class DecimalTableExample(Scene):
    def construct(self):
        x_vals = [-2,-1,0,1,2]
        y_vals = np.exp(x_vals)
        t0 = DecimalTable(
            [x_vals, y_vals],
            row_labels=[MathTex("x"), MathTex("f(x)=e^{x}")],
            h_buff=1,
            element_to_mobject_config={"num_decimal_places": 2})

```

```
        self.add(t0)
"""

def __init__(  
    self,  
    table: Iterable[Iterable[float | str]],  
    element_to_mobject: Callable[[float | str], VMobject] = DecimalNumber,  
    element_to_mobject_config: dict = {"num_decimal_places": 1},  
    **kwargs,  
):  
    """  
    Special case of :class:`~.Table` with ``element_to_mobject`` set to :class:`~.DecimalNumber`.  
    By default, ``num_decimal_places`` is set to 1.  
    Will round/truncate the decimal places based on the provided ``element_to_mobject_config``.  
  
    Parameters  
    -----  
    table  
        A 2D array, or a list of lists. Content of the table must be valid input  
        for :class:`~.DecimalNumber`.  
    element_to_mobject  
        The :class:`~.Mobject` class applied to the table entries. Set as :class:`~.DecimalNumber`.  
    element_to_mobject_config  
        Element to mobject config, here set as {"num_decimal_places": 1}.  
    kwargs  
        Additional arguments to be passed to :class:`~.Table`.  
    """  
    super().__init__(  
        table,  
        element_to_mobject=element_to_mobject,  
        element_to_mobject_config=element_to_mobject_config,  
        **kwargs,  
    )
```

probability.py

```
"""Mobjects representing objects from probability theory and statistics."""
from __future__ import annotations
__all__ = ["SampleSpace", "BarChart"]

from typing import Iterable, MutableSequence, Sequence

import numpy as np
from colour import Color

from manim import config, logger
from manim.constants import *
from manim.mobject.geometry.polygram import Rectangle
from manim.mobject.graphing.coordinate_systems import Axes
from manim.mobject.mobject import Mobject
from manim.mobject.opengl.opengl_mobject import OpenGLObject
from manim.mobject.svg.brace import Brace
from manim.mobject.text.tex_mobject import MathTex, Tex
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.color import (
    BLUE_E,
    DARK_GREY,
    GREEN_E,
    LIGHT_GREY,
    MAROON_B,
    YELLOW,
    color_gradient,
)
from manim.utils.iterables import tuplify

EPSILON = 0.0001

class SampleSpace(Rectangle):
    """A mobject representing a twodimensional rectangular sampling space.

    Examples
    -----
    .. manim:: ExampleSampleSpace
       :save_last_frame:

    class ExampleSampleSpace(Scene):
        def construct(self):
            poly1 = SampleSpace(stroke_width=15, fill_opacity=1)
            poly2 = SampleSpace(width=5, height=3, stroke_width=5, fill_opacity=0.5)
            poly3 = SampleSpace(width=2, height=2, stroke_width=5, fill_opacity=0.1)
            poly3.divide_vertically(p_list=np.array([0.37, 0.13, 0.5]), colors=[BLACK, WHITE, GRAY], vect=RIGHT)
            poly_group = VGroup(poly1, poly2, poly3).arrange()
            self.add(poly_group)
    """

    def __init__(self,
                 height=3,
                 width=3,
                 fill_color=DARK_GREY,
                 fill_opacity=1,
                 stroke_width=0.5,
                 stroke_color=LIGHT_GREY,
                 default_label_scale_val=1,
                 ):
        super().__init__(
            height=height,
            width=width,
            fill_color=fill_color,
            fill_opacity=fill_opacity,
            stroke_width=stroke_width,
            stroke_color=stroke_color,
        )
        self.default_label_scale_val = default_label_scale_val

    def add_title(self, title="Sample space", buff=MED_SMALL_BUFF):
        # TODO, should this really exist in SampleSpaceScene
        title_mob = Tex(title)
        if title_mob.width > self.width:
            title_mob.width = self.width
        title_mob.next_to(self, UP, buff=buff)
        self.title = title_mob
        self.add(title_mob)

    def add_label(self, label):
        self.label = label

    def complete_p_list(self, p_list):
        new_p_list = list(tuplify(p_list))
        remainder = 1.0 - sum(new_p_list)
        if abs(remainder) > EPSILON:
            new_p_list.append(remainder)
        return new_p_list

    def get_division_along_dimension(self, p_list, dim, colors, vect):
        p_list = self.complete_p_list(p_list)
        colors = color_gradient(colors, len(p_list))

        last_point = self.get_edge_center(-vect)
        parts = VGroup()
        for factor, color in zip(p_list, colors):
            part = SampleSpace()
            part.set_fill(color, 1)
            part.replace(self, stretch=True)
            part.stretch(factor, dim)
            part.move_to(last_point, -vect)
            last_point = part.get_edge_center(vect)
            parts.add(part)
        return parts
```

```

def get_horizontal_division(self, p_list, colors=[GREEN_E, BLUE_E], vect=DOWN):
    return self.get_division_along_dimension(p_list, 1, colors, vect)

def get_vertical_division(self, p_list, colors=[MAROON_B, YELLOW], vect=RIGHT):
    return self.get_division_along_dimension(p_list, 0, colors, vect)

def divide_horizontally(self, *args, **kwargs):
    self.horizontal_parts = self.get_horizontal_division(*args, **kwargs)
    self.add(self.horizontal_parts)

def divide_vertically(self, *args, **kwargs):
    self.vertical_parts = self.get_vertical_division(*args, **kwargs)
    self.add(self.vertical_parts)

def get_subdivision_braces_and_labels(
    self,
    parts,
    labels,
    direction,
    buff=SMALL_BUFF,
    min_num_quads=1,
):
    label_mobs = VGroup()
    braces = VGroup()
    for label, part in zip(labels, parts):
        brace = Brace(part, direction, min_num_quads=min_num_quads, buff=buff)
        if isinstance(label, (Mobject, OpenGLMobject)):
            label_mob = label
        else:
            label_mob = MathTex(label)
            label_mob.scale(self.default_label_scale_val)
        label_mob.next_to(brace, direction, buff)

        braces.add(brace)
        label_mobs.add(label_mob)
    parts.braces = braces
    parts.labels = label_mobs
    parts.label_kw_args = {
        "labels": label_mobs.copy(),
        "direction": direction,
        "buff": buff,
    }
    return VGroup(parts.braces, parts.labels)

def get_side_braces_and_labels(self, labels, direction=LEFT, **kwargs):
    assert hasattr(self, "horizontal_parts")
    parts = self.horizontal_parts
    return self.get_subdivision_braces_and_labels(
        parts, labels, direction, **kwargs
    )

def get_top_braces_and_labels(self, labels, **kwargs):
    assert hasattr(self, "vertical_parts")
    parts = self.vertical_parts
    return self.get_subdivision_braces_and_labels(parts, labels, UP, **kwargs)

def get_bottom_braces_and_labels(self, labels, **kwargs):
    assert hasattr(self, "vertical_parts")
    parts = self.vertical_parts
    return self.get_subdivision_braces_and_labels(parts, labels, DOWN, **kwargs)

def add_braces_and_labels(self):
    for attr in "horizontal_parts", "vertical_parts":
        if not hasattr(self, attr):
            continue
        parts = getattr(self, attr)
        for subattr in "braces", "labels":
            if hasattr(parts, subattr):
                self.add(getattr(parts, subattr))

def __getitem__(self, index):
    if hasattr(self, "horizontal_parts"):
        return self.horizontal_parts[index]
    elif hasattr(self, "vertical_parts"):
        return self.vertical_parts[index]
    return self.split()[index]

class BarChart(Axes):
    """Creates a bar chart. Inherits from :class:`~.Axes`, so it shares its methods and attributes. Each axis inherits from :class:`~.NumberLine`, so pass in ``x_axis_config``/``y_axis_config`` to control their attributes.

    Parameters
    -----
    values
        A sequence of values that determines the height of each bar. Accepts negative values.
    bar_names
        A sequence of names for each bar. Does not have to match the length of ``values``.
    y_range
        The y-axis range of values. If ``None``, the range will be calculated based on the min/max of ``values`` and the step will be calculated based on ``y_length``.
    x_length
        The length of the x-axis. If ``None``, it is automatically calculated based on the number of values and the width of the screen.
    y_length
        The length of the y-axis.
    bar_colors
        The color for the bars. Accepts a sequence of colors (can contain just one item). If the length of ``bar_colors`` does not match that of ``values``, intermediate colors will be automatically determined.
    bar_width
        The length of a bar. Must be between 0 and 1.
    bar_fill_opacity
        The fill opacity of the bars.
    bar_stroke_width
        The stroke width of the bars.

    Examples
    -----
    .. manim:: BarChartExample
       :save_last_frame:

       class BarChartExample(Scene):
           def construct(self):

```

```

chart = BarChart(
    values=[-5, 40, -10, 20, -3],
    bar_names=["one", "two", "three", "four", "five"],
    y_range=[-20, 50, 10],
    y_length=6,
    x_length=10,
    x_axis_config={"font_size": 36},
)
c_bar_lbls = chart.get_bar_labels(font_size=48)
self.add(chart, c_bar_lbls)

"""
def __init__(
    self,
    values: MutableSequence[float],
    bar_names: Sequence[str] | None = None,
    y_range: Sequence[float] | None = None,
    x_length: float | None = None,
    y_length: float | None = None,
    bar_colors: Iterable[str] = [
        "#0035C",
        "#58508d",
        "#bc5090",
        "#ff6361",
        "#ffa600",
    ],
    bar_width: float = 0.6,
    bar_fill_opacity: float = 0.7,
    bar_stroke_width: float = 3,
    **kwargs,
):
    if isinstance(bar_colors, str):
        logger.warning(
            "Passing a string to `bar_colors` has been deprecated since v0.15.2 and will be removed after v0.17.0, the parameter must be a list. "
        )
        bar_colors = list(bar_colors)
    y_length = y_length if y_length is not None else config.frame_height - 4
    self.values = values
    self.bar_names = bar_names
    self.bar_colors = bar_colors
    self.bar_width = bar_width
    self.bar_fill_opacity = bar_fill_opacity
    self.bar_stroke_width = bar_stroke_width
    x_range = [0, len(self.values), 1]
    if y_range is None:
        y_range = [
            min(0, min(self.values)),
            max(0, max(self.values)),
            round(max(self.values) / y_length, 2),
        ]
    elif len(y_range) == 2:
        y_range = [*y_range, round(max(self.values) / y_length, 2)]
    if x_length is None:
        x_length = min(len(self.values), config.frame_width - 2)
    x_axis_config = {"font_size": 24, "label_constructor": Tex}
    self._update_default_configs(
        (x_axis_config,), (kwargs.pop("x_axis_config", None),)
    )
    self.bars: VGroup = VGroup()
    self.x_labels: VGroup | None = None
    self.bar_labels: VGroup | None = None
    super().__init__(
        x_range=x_range,
        y_range=y_range,
        x_length=x_length,
        y_length=y_length,
        x_axis_config=x_axis_config,
        tips=kwargs.pop("tips", False),
        **kwargs,
    )
    self._add_bars()
    if self.bar_names is not None:
        self._add_x_axis_labels()
    self.y_axis.add_numbers()
def _update_colors(self):
    """Initialize the colors of the bars of the chart.

    Sets the color of ``self.bars`` via ``self.bar_colors``.

    Primarily used when the bars are initialized with ``self._add_bars``
    or updated via ``self.change_bar_values``.
    """
    self.bars.set_color_by_gradient(*self.bar_colors)
def _add_x_axis_labels(self):
    """Essentially :meth:`~.NumberLine.add_labels`, but differs in that
    the direction of the label with respect to the x_axis changes to UP or DOWN
    depending on the value.

    UP for negative values and DOWN for positive values.
    """
    val_range = np.arange(
        0.5, len(self.bar_names), 1
    ) # 0.5 shifted so that labels are centered, not on ticks
    labels = VGroup()
    for i, (value, bar_name) in enumerate(zip(val_range, self.bar_names)):

```

```

# to accommodate negative bars, the label may need to be
# below or above the x_axis depending on the value of the bar
if self.values[i] < 0:
    direction = UP
else:
    direction = DOWN
bar_name_label = self.x_axis.label_constructor(bar_name)

bar_name_label.font_size = self.x_axis.font_size
bar_name_label.next_to(
    self.x_axis.number_to_point(value),
    direction=direction,
    buff=self.x_axis.line_to_number_buff,
)
labels.add(bar_name_label)

self.x_axis.labels = labels
self.x_axis.add(labels)

def create_bar(self, bar_number: int, value: float) -> Rectangle:
    """Creates a positioned bar on the chart.

    Parameters
    -----
    bar_number
        Determines the x-position of the bar.
    value
        The value that determines the height of the bar.

    Returns
    -----
    Rectangle
        A positioned rectangle representing a bar on the chart.
    """

    # bar measurements relative to the axis

    # distance from between the y-axis and the top of the bar
    bar_h = abs(self.c2p(0, value)[1] - self.c2p(0, 0)[1])
    # width of the bar
    bar_w = self.c2p(self.bar_width, 0)[0] - self.c2p(0, 0)[0]

    bar = Rectangle(
        height=bar_h,
        width=bar_w,
        stroke_width=self.bar_stroke_width,
        fill_opacity=self.bar_fill_opacity,
    )

    pos = UP if (value >= 0) else DOWN
    bar.next_to(self.c2p(bar_number + 0.5, 0), pos, buff=0)
    return bar

def _add_bars(self) -> None:
    for i, value in enumerate(self.values):
        tmp_bar = self._create_bar(bar_number=i, value=value)
        self.bars.add(tmp_bar)

    self._update_colors()
    self.add_to_back(self.bars)

def get_bar_labels(
    self,
    color: Color | None = None,
    font_size: float = 24,
    buff: float = MED_SMALL_BUFF,
    label_constructor: type[VMobject] = Tex,
):
    """Annotates each bar with its corresponding value. Use ``self.bar_labels`` to access the
    labels after creation.

    Parameters
    -----
    color
        The color of each label. By default ``None`` and is based on the parent's bar color.
    font_size
        The font size of each label.
    buff
        The distance from each label to its bar. By default 0.4.
    label_constructor
        The Mobject class to construct the labels, by default :class:`~.Tex`.

    Examples
    -----
    .. manim:: GetBarLabelsExample
        :save_last_frame:

        class GetBarLabelsExample(Scene):
            def construct(self):
                chart = BarChart(values=[10, 9, 8, 7, 6, 5, 4, 3, 2, 1], y_range=[0, 10, 1])
                c_bar_lbls = chart.get_bar_labels(
                    color=WHITE, label_constructor=MathTex, font_size=36
                )
                self.add(chart, c_bar_lbls)
    """

    bar_labels = VGroup()
    for bar, value in zip(self.bars, self.values):
        bar_lbl = label_constructor(str(value))

        if color is None:
            bar_lbl.set_color(bar.get_fill_color())
        else:
            bar_lbl.set_color(color)

        bar_lbl.font_size = font_size

        pos = UP if (value >= 0) else DOWN
        bar_lbl.next_to(bar, pos, buff=buff)
        bar_labels.add(bar_lbl)

    return bar_labels

```

```

def change_bar_values(self, values: Iterable[float], update_colors: bool = True):
    """Updates the height of the bars of the chart.

Parameters
-----
values
    The values that will be used to update the height of the bars.
    Does not have to match the number of bars.
update_colors
    Whether to re-initialize the colors of the bars based on ``self.bar_colors``.

Examples
-----
.. manim:: ChangeBarValuesExample
    :save_last_frame:

    class ChangeBarValuesExample(Scene):
        def construct(self):
            values=[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]

            chart = BarChart(
                values,
                y_range=[-10, 10, 2],
                y_axis_config={"font_size": 24},
            )
            self.add(chart)

            chart.change_bar_values(list(reversed(values)))
            self.add(chart.get_bar_labels(font_size=24))
    """

for i, (bar, value) in enumerate(zip(self.bars, values)):
    chart_val = self.values[i]

    if chart_val > 0:
        bar_lim = bar.get_bottom()
        aligned_edge = DOWN
    else:
        bar_lim = bar.get_top()
        aligned_edge = UP

    # check if the bar has height
    if chart_val != 0:
        quotient = value / chart_val
        if quotient < 0:
            aligned_edge = UP if chart_val > 0 else DOWN

        # if the bar is already positive, then we now want to move it
        # so that it is negative. So, we move the top edge of the bar
        # to the location of the previous bottom

        # if already negative, then we move the bottom edge of the bar
        # to the location of the previous top

        bar.stretch_to_fit_height(abs(quotient) * bar.height)

    else:
        # create a new bar since the current one has a height of zero (doesn't exist)
        temp_bar = self._create_bar(i, value)
        self.bars.remove(bar)
        self.bars.insert(i, temp_bar)

    bar.move_to(bar_lim, aligned_edge)

if update_colors:
    self._update_colors()

self.values[: len(values)] = values

```

three_d_camera.py

```
"""A camera that can be positioned and oriented in three-dimensional space."""
from __future__ import annotations
__all__ = ["ThreeDCamera"]

from typing import Callable
import numpy as np

from manim.mobject.mobject import Mobject
from manim.mobject.three_d.three_d_utils import (
    get_3d_vmob_end_corner,
    get_3d_vmob_end_corner_unit_normal,
    get_3d_vmob_start_corner,
    get_3d_vmob_start_corner_unit_normal,
)
from manim.mobject.value_tracker import ValueTracker

from .. import config
from ..camera.camera import Camera
from ..constants import *
from ..mobject.types.point_cloud_mobject import Point
from ..utils.color import get_shaded_rgb
from ..utils.family import extract_mobject_family_members
from ..utils.space_ops import rotation_about_z, rotation_matrix

class ThreeDCamera(Camera):
    def __init__(
        self,
        focal_distance=20.0,
        shading_factor=0.2,
        default_distance=5.0,
        light_source_start_point=9 * DOWN + 7 * LEFT + 10 * OUT,
        should_apply_shading=True,
        exponential_projection=False,
        phi=0,
        theta=-90 * DEGREES,
        gamma=0,
        zoom=1,
        **kwargs,
    ):
        """Initializes the ThreeDCamera

        Parameters
        -----
        *kwargs
            Any keyword argument of Camera.

        """
        self._frame_center = Point(kwargs.get("frame_center", ORIGIN), stroke_width=0)
        super().__init__(**kwargs)
        self.focal_distance = focal_distance
        self.phi = phi
        self.theta = theta
        self.gamma = gamma
        self.zoom = zoom
        self.shading_factor = shading_factor
        self.default_distance = default_distance
        self.light_source_start_point = light_source_start_point
        self.light_source = Point(self.light_source_start_point)
        self.should_apply_shading = should_apply_shading
        self.exponential_projection = exponential_projection
        self.max_allowable_norm = 3 * config["frame_width"]
        self.phi_tracker = ValueTracker(self.phi)
        self.theta_tracker = ValueTracker(self.theta)
        self.focal_distance_tracker = ValueTracker(self.focal_distance)
        self.gamma_tracker = ValueTracker(self.gamma)
        self.zoom_tracker = ValueTracker(self.zoom)
        self.fixed_orientation_mobjects = {}
        self.fixed_in_frame_mobjects = set()
        self.reset_rotation_matrix()

    @property
    def frame_center(self):
        return self._frame_center.points[0]

    @frame_center.setter
    def frame_center(self, point):
        self._frame_center.move_to(point)
```

```

def capture_mobjects(self, mobjects, **kwargs):
    self.reset_rotation_matrix()
    super().capture_mobjects(mobjects, **kwargs)

def get_value_trackers(self):
    """A list of :class:`ValueTrackers` of phi, theta, focal_distance,
    gamma and zoom.

    Returns
    ------
    list
        list of ValueTracker objects
    """
    return [
        self.phi_tracker,
        self.theta_tracker,
        self.focal_distance_tracker,
        self.gamma_tracker,
        self.zoom_tracker,
    ]

def modified_rgbs(self, vmobject, rgbs):
    if not self.should_apply_shading:
        return rgbs
    if vmobject.shade_in_3d and (vmobject.get_num_points() > 0):
        light_source_point = self.light_source.points[0]
        if len(rgbs) < 2:
            shaded_rgbs = rgbs.repeat(2, axis=0)
        else:
            shaded_rgbs = np.array(rgbs[:2])
            shaded_rgbs[0, :3] = get_shaded_rgb(
                shaded_rgbs[0, :3],
                get_3d_vmob_start_corner(vmobject),
                get_3d_vmob_start_corner_unit_normal(vmobject),
                light_source_point,
            )
            shaded_rgbs[1, :3] = get_shaded_rgb(
                shaded_rgbs[1, :3],
                get_3d_vmob_end_corner(vmobject),
                get_3d_vmob_end_corner_unit_normal(vmobject),
                light_source_point,
            )
        return shaded_rgbs
    return rgbs

def get_stroke_rgbs(
    self,
    vmobject,
    background=False,
):
    # NOTE : DocStrings From parent
    return self.modified_rgbs(vmobject, vmobject.get_stroke_rgbs(background))

def get_fill_rgbs(self, vmobject): # NOTE : DocStrings From parent
    return self.modified_rgbs(vmobject, vmobject.get_fill_rgbs())

def get_mobjects_to_display(self, *args, **kwargs): # NOTE : DocStrings From parent
    mobjects = super().get_mobjects_to_display(*args, **kwargs)
    rot_matrix = self.get_rotation_matrix()

    def z_key(mob):
        if not (hasattr(mob, "shade_in_3d") and mob.shade_in_3d):
            return np.inf
        # Assign a number to a three dimensional mobjects
        # based on how close it is to the camera
        return np.dot(mob.get_z_index_reference_point(), rot_matrix.T)[2]

    return sorted(mobjects, key=z_key)

def get_phi(self):
    """Returns the Polar angle (the angle off Z_AXIS) phi.

    Returns
    ------
    float
        The Polar angle in radians.
    """
    return self.phi_tracker.get_value()

def get_theta(self):
    """Returns the Azimuthal i.e the angle that spins the camera around the Z_AXIS.

    Returns
    ------
    float
        The Azimuthal angle in radians.
    """

```

```

    return self.theta_tracker.get_value()

def get_focal_distance(self):
    """Returns focal_distance of the Camera.

    Returns
    ------
    float
        The focal_distance of the Camera in MUnits.
    """
    return self.focal_distance_tracker.get_value()

def get_gamma(self):
    """Returns the rotation of the camera about the vector from the ORIGIN to the Camera.

    Returns
    ------
    float
        The angle of rotation of the camera about the vector
        from the ORIGIN to the Camera in radians
    """
    return self.gamma_tracker.get_value()

def get_zoom(self):
    """Returns the zoom amount of the camera.

    Returns
    ------
    float
        The zoom amount of the camera.
    """
    return self.zoom_tracker.get_value()

def set_phi(self, value: float):
    """Sets the polar angle i.e the angle between Z_AXIS and Camera through ORIGIN in radians.

    Parameters
    ------
    value
        The new value of the polar angle in radians.
    """
    self.phi_tracker.set_value(value)

def set_theta(self, value: float):
    """Sets the azimuthal angle i.e the angle that spins the camera around Z_AXIS in radians.

    Parameters
    ------
    value
        The new value of the azimuthal angle in radians.
    """
    self.theta_tracker.set_value(value)

def set_focal_distance(self, value: float):
    """Sets the focal_distance of the Camera.

    Parameters
    ------
    value
        The focal_distance of the Camera.
    """
    self.focal_distance_tracker.set_value(value)

def set_gamma(self, value: float):
    """Sets the angle of rotation of the camera about the vector from the ORIGIN to the Camera.

    Parameters
    ------
    value
        The new angle of rotation of the camera.
    """
    self.gamma_tracker.set_value(value)

def set_zoom(self, value: float):
    """Sets the zoom amount of the camera.

    Parameters
    ------
    value
        The zoom amount of the camera.
    """
    self.zoom_tracker.set_value(value)

def reset_rotation_matrix(self):
    """Sets the value of self.rotation_matrix to
    the matrix corresponding to the current position of the camera
    """

```

```

    self.rotation_matrix = self.generate_rotation_matrix()

def get_rotation_matrix(self):
    """Returns the matrix corresponding to the current position of the camera.

    Returns
    ------
    np.array
        The matrix corresponding to the current position of the camera.
    """
    return self.rotation_matrix

def generate_rotation_matrix(self):
    """Generates a rotation matrix based off the current position of the camera.

    Returns
    ------
    np.array
        The matrix corresponding to the current position of the camera.
    """
    phi = self.get_phi()
    theta = self.get_theta()
    gamma = self.get_gamma()
    matrices = [
        rotation_about_z(-theta - 90 * DEGREES),
        rotation_matrix(-phi, RIGHT),
        rotation_about_z(gamma),
    ]
    result = np.identity(3)
    for matrix in matrices:
        result = np.dot(matrix, result)
    return result

def project_points(self, points: np.ndarray | list):
    """Applies the current rotation_matrix as a projection
    matrix to the passed array of points.

    Parameters
    -----
    points
        The list of points to project.

    Returns
    ------
    np.array
        The points after projecting.
    """
    frame_center = self.frame_center
    focal_distance = self.get_focal_distance()
    zoom = self.get_zoom()
    rot_matrix = self.get_rotation_matrix()

    points = points - frame_center
    points = np.dot(points, rot_matrix.T)
    zs = points[:, 2]
    for i in 0, 1:
        if self.exponential_projection:
            # Proper projection would involve multiplying
            # x and y by d / (d-z). But for points with high
            # z value that causes weird artifacts, and applying
            # the exponential helps smooth it out.
            factor = np.exp(zs / focal_distance)
            lt0 = zs < 0
            factor[lt0] = focal_distance / (focal_distance - zs[lt0])
        else:
            factor = focal_distance / (focal_distance - zs)
            factor[(focal_distance - zs) < 0] = 10**6
        points[:, i] *= factor * zoom
    return points

def project_point(self, point: list | np.ndarray):
    """Applies the current rotation_matrix as a projection
    matrix to the passed point.

    Parameters
    -----
    point
        The point to project.

    Returns
    ------
    np.array
        The point after projection.
    """
    return self.project_points(point.reshape((1, 3)))[0, :]

def transform_points_pre_display(

```

```

self,
mobject,
points,
): # TODO: Write Docstrings for this Method.
points = super().transform_points_pre_display(mobject, points)
fixed_orientation = mobject in self.fixed_orientation_mobjects
fixed_in_frame = mobject in self.fixed_in_frame_mobjects

if fixed_in_frame:
    return points
if fixed_orientation:
    center_func = self.fixed_orientation_mobjects[mobject]
    center = center_func()
    new_center = self.project_point(center)
    return points + (new_center - center)
else:
    return self.project_points(points)

def add_fixed_orientation_mobjects(
    self,
    *mobjects: Mobject,
    use_static_center_func: bool = False,
    center_func: Callable[[], np.ndarray] | None = None,
):
    """This method allows the mobject to have a fixed orientation,
    even when the camera moves around.
    E.G If it was passed through this method, facing the camera, it
    will continue to face the camera even as the camera moves.
    Highly useful when adding labels to graphs and the like.

    Parameters
    -----
    *mobjects
        The mobject whose orientation must be fixed.
    use_static_center_func
        Whether or not to use the function that takes the mobject's
        center as centerpoint, by default False
    center_func
        The function which returns the centerpoint
        with respect to which the mobject will be oriented, by default None
    """

    # This prevents the computation of mobject.get_center
    # every single time a projection happens
    def get_static_center_func(mobject):
        point = mobject.get_center()
        return lambda: point

    for mobject in mobjects:
        if center_func:
            func = center_func
        elif use_static_center_func:
            func = get_static_center_func(mobject)
        else:
            func = mobject.get_center
        for submob in mobject.get_family():
            self.fixed_orientation_mobjects[submob] = func

def add_fixed_in_frame_mobjects(self, *mobjects: Mobject):
    """This method allows the mobject to have a fixed position,
    even when the camera moves around.
    E.G If it was passed through this method, at the top of the frame, it
    will continue to be displayed at the top of the frame.

    Highly useful when displaying Titles or formulae or the like.

    Parameters
    -----
    **mobjects
        The mobject to fix in frame.
    """

    for mobject in extract_mobject_family_members(mobjects):
        self.fixed_in_frame_mobjects.add(mobject)

def remove_fixed_orientation_mobjects(self, *mobjects: Mobject):
    """If a mobject was fixed in its orientation by passing it through
    :meth:`.add_fixed_orientation_mobjects`, then this undoes that fixing.
    The Mobject will no longer have a fixed orientation.

    Parameters
    -----
    mobjects
        The mobjects whose orientation need not be fixed any longer.
    """

    for mobject in extract_mobject_family_members(mobjects):
        if mobject in self.fixed_orientation_mobjects:
            del self.fixed_orientation_mobjects[mobject]

```

```
def remove_fixed_in_frame_mobjects(self, *mobjects: Mobject):
    """If a mobject was fixed in frame by passing it through
    :meth:`.add_fixed_in_frame_mobjects`, then this undoes that fixing.
    The Mobject will no longer be fixed in frame.

    Parameters
    -----
    mobjects
        The mobjects which need not be fixed in frame any longer.
    """
    for mobject in extract_mobject_family_members(mobjects):
        if mobject in self.fixed_in_frame_mobjects:
            self.fixed_in_frame_mobjects.remove(mobject)
```

tex_templates.py

```
"""A library of LaTeX templates."""
from __future__ import annotations

__all__ = [
    "TexTemplateLibrary",
    "TexFontTemplates",
]
from .tex import *

# This file makes TexTemplateLibrary and TexFontTemplates available for use in manim Tex and MathTex objects.

def _new_ams_template():
    """Returns a simple Tex Template with only basic AMS packages"""
    preamble = r"""
\usepackage[english]{babel}
\usepackage{amsmath}
\usepackage{amssymb}
"""
    return TexTemplate(preamble=preamble)

""" Tex Template preamble used by original upstream 3b1b """
_3b1b_preamble = r"""
\usepackage[english]{babel}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\usepackage{amsmath}
\usepackage{amssymb}
\usepackage{dsfont}
\usepackage{setspace}
\usepackage{tipa}
\usepackage{relsize}
\usepackage{textcomp}
\usepackage{mathrsfs}
\usepackage{calligra}
\usepackage{wasysym}
\usepackage{ragged2e}
\usepackage{physics}
\usepackage{xcolor}
\usepackage{microtype}
\DisableLigatures{encoding = *, family = * }
\linespread{1}
"""

# TexTemplateLibrary
#
class TexTemplateLibrary:
    """
    A collection of basic TeX template objects

    Examples
    -----
    Normal usage as a value for the keyword argument tex_template of Tex() and MathTex() mobjects::

        ``Tex("My TeX code", tex_template=TexTemplateLibrary.ctex)``
    """

    default = TexTemplate(preamble=_3b1b_preamble)
    """An instance of the default TeX template in manim"""

    threeb1b = TexTemplate(preamble=_3b1b_preamble)
    """ An instance of the default TeX template used by 3b1b """

    ctex = TexTemplate(
        tex_compiler="xelatex",
        output_format=".xdv",
        preamble=_3b1b_preamble.replace(
            r"\DisableLigatures{encoding = *, family = * }",
            r"\usepackage[UTF8]{ctex}",
        ),
    )
    """An instance of the TeX template used by 3b1b when using the use_ctex flag"""

    simple = _new_ams_template()
    """An instance of a simple TeX template with only basic AMS packages loaded"""
```



```

\usepackage[T1]{fontenc}
\usepackage[variablett]{lmodern}
\renewcommand{\rmdefault}{\ttdefault}
\usepackage[LGRgreek]{mathastext}
\MTgreekfont{lmtt} % no lgr lmvt, so use lgr lmtt
\Mathastext
\let\varepsilon\epsilon % only \varsigma in LGR
"",
)

# Fourier Utopia (Fourier upright Greek)
fufug = _new_ams_template()
fufug.description = "Fourier Utopia (Fourier upright Greek)"
fufug.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[upright]{fourier}
\usepackage{mathastext}
""",
)

# Droid Serif
droidserif = _new_ams_template()
droidserif.description = "Droid Serif"
droidserif.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[default]{droidserif}
\usepackage[LGRgreek]{mathastext}
\let\varepsilon\epsilon
""",
)

# Droid Sans
droidsans = _new_ams_template()
droidsans.description = "Droid Sans"
droidsans.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[default]{droidsans}
\usepackage[LGRgreek]{mathastext}
\let\varepsilon\epsilon
""",
)

# New Century Schoolbook (Symbol Greek)
ncssg = _new_ams_template()
ncssg.description = "New Century Schoolbook (Symbol Greek)"
ncssg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{newcent}
\usepackage[symbolgreek]{mathastext}
\linespread{1.1}
""",
)

# French Cursive (Euler Greek)
fceg = _new_ams_template()
fceg.description = "French Cursive (Euler Greek)"
fceg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[default]{frcursive}
\usepackage[eulergreek,noplusnominus,noequal,nohbar,%nolessnomore,noasterisk]{mathastext}
""",
)

# Auriocus Kalligraphicus (Symbol Greek)
aksg = _new_ams_template()
aksg.description = "Auriocus Kalligraphicus (Symbol Greek)"
aksg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[aurical]
\renewcommand{\rmdefault}{AuriocusKalligraphicus}
\usepackage[symbolgreek]{mathastext}
""",
)

```

```

# Palatino (Symbol Greek)
palatinosg = _new_ams_template()
palatinosg.description = "Palatino (Symbol Greek)"
palatinosg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{palatino}
\usepackage[symbolmax,defaultmathsizes]{mathastext}
""",
)

# Comfortaa
comfortaa = _new_ams_template()
comfortaa.description = "Comfortaa"
comfortaa.add_to_preamble(
    r"""
\usepackage[default]{comfortaa}
\usepackage[LGRgreek,defaultmathsizes,noasterisk]{mathastext}
\let\varphi\phi
\linespread{1.06}
""",
)

# ECF Augie (Euler Greek)
ecfaugieeg = _new_ams_template()
ecfaugieeg.description = "ECF Augie (Euler Greek)"
ecfaugieeg.add_to_preamble(
    r"""
\renewcommand\familydefault\fau % emerald package
\usepackage[defaultmathsizes,eulergreek]{mathastext}
""",
)

# Electrum ADF (CM Greek)
electrumadfcml = _new_ams_template()
electrumadfcml.description = "Electrum ADF (CM Greek)"
electrumadfcml.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[LGRgreek,basic,defaultmathsizes]{mathastext}
\usepackage[lf]{electrum}
\Mathastext
\let\varphi\phi
""",
)

# American Typewriter
americantypewriter = _new_ams_template()
americantypewriter.description = "American Typewriter"
americantypewriter.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{American Typewriter}
\usepackage[defaultmathsizes]{mathastext}
""",
)
americantypewriter.tex_compiler = "xelatex"
americantypewriter.output_format = ".xdv"

# Minion Pro and Myriad Pro (and TX fonts symbols)
mpmptx = _new_ams_template()
mpmptx.description = "Minion Pro and Myriad Pro (and TX fonts symbols)"
mpmptx.add_to_preamble(
    r"""
\usepackage{txfonts}
\usepackage[upright]{txgreeks}
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Minion Pro}
\setsansfont[Mapping=tex-text,Scale=MatchUppercase]{Myriad Pro}
\renewcommand\familydefault\sfdefault
\usepackage[defaultmathsizes]{mathastext}
\renewcommand\familydefault\rmdefault
""",
)
mpmptx.tex_compiler = "xelatex"
mpmptx.output_format = ".xdv"

# New Century Schoolbook (Symbol Greek, PX math symbols)
ncssgpxm = _new_ams_template()
ncssgpxm.description = "New Century Schoolbook (Symbol Greek, PX math symbols)"
ncssgpxm.add_to_preamble(

```

```

r"""
\usepackage[T1]{fontenc}
\usepackage{pxfonts}
\usepackage{newcent}
\usepackage[symbolgreek,defaultmathsizes]{mathastext}
\linespread{1.06}
""",
)

# Vollkorn (TX fonts for Greek and math symbols)
vollkorntx = _new_ams_template()
vollkorntx.description = "Vollkorn (TX fonts for Greek and math symbols)"
vollkorntx.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{txfonts}
\usepackage[upright]{txgreeks}
\usepackage{vollkorn}
\usepackage[defaultmathsizes]{mathastext}
""",
)

```

```

# Libertine
libertine = _new_ams_template()
libertine.description = "Libertine"
libertine.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{libertine}
\usepackage[greek=n]{libgreek}
\usepackage[noasterisk,defaultmathsizes]{mathastext}
""",
)

```

```

# SliTeX (Euler Greek)
slitexeg = _new_ams_template()
slitexeg.description = "SliTeX (Euler Greek)"
slitexeg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{tpslifonts}
\usepackage[eulergreek,defaultmathsizes]{mathastext}
\MTEulerScale{1.06}
\linespread{1.2}
""",
)

```

```

# ECF Webster (with TX fonts)
ecfwebstertx = _new_ams_template()
ecfwebstertx.description = "ECF Webster (with TX fonts)"
ecfwebstertx.add_to_preamble(
    r"""
\usepackage{txfonts}
\usepackage[upright]{txgreeks}
\renewcommand\familydefault\sfdefault % emerald package
\usepackage{mathastext}
\renewcommand{\int}{\intop\limits}
\linespread{1.5}
""",
)
ecfwebstertx.add_to_document(
    r"""
\mathversion{bold}
""",
)

```

```

# Romande ADF with Fourier (Italic)
italicromandeadff = _new_ams_template()
italicromandeadff.description = "Romande ADF with Fourier (Italic)"
italicromandeadff.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{fourier}
\usepackage{romande}
\usepackage[italic,defaultmathsizes,noasterisk]{mathastext}
\renewcommand{\itshape}{\swashstyle}
""",
)

```

```

# Apple Chancery
applechancery = _new_ams_template()

```

```

applechancery.description = "Apple Chancery"
applechancery.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Apple Chancery}
\usepackage[defaultmathsizes]{mathastext}
""",
)
applechancery.tex_compiler = "xelatex"
applechancery.output_format = ".xdv"

# Zapf Chancery
zapfchancery = _new_ams_template()
zapfchancery.description = "Zapf Chancery"
zapfchancery.add_to_preamble(
    r"""
\DeclareFontFamily{T1}{pzc} {}
\DeclareFontShape{T1}{pzc}{mb}{it}{<->s*[1.2] pzcmi8t} {}
\DeclareFontShape{T1}{pzc}{m}{it}{<->ssub * pzc/mb/it} {}
\usepackage{chancery} % = \renewcommand{\rmdefault}{pzc}
\renewcommand\shapedefault\itdefault
\renewcommand\bfdefault\mddefault
\usepackage[defaultmathsizes]{mathastext}
\linespread{1.05}
""",
)
)

# Verdana (Italic)
italicverdana = _new_ams_template()
italicverdana.description = "Verdana (Italic)"
italicverdana.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Verdana}
\usepackage[defaultmathsizes,italic]{mathastext}
""",
)
italicverdana.tex_compiler = "xelatex"
italicverdana.output_format = ".xdv"

# URW Zapf Chancery (CM Greek)
urwzccmg = _new_ams_template()
urwzccmg.description = "URW Zapf Chancery (CM Greek)"
urwzccmg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\DeclareFontFamily{T1}{pzc} {}
\DeclareFontShape{T1}{pzc}{mb}{it}{<->s*[1.2] pzcmi8t} {}
\DeclareFontShape{T1}{pzc}{m}{it}{<->ssub * pzc/mb/it} {}
\DeclareFontShape{T1}{pzc}{mb}{sl}{<->ssub * pzc/mb/it} {}
\DeclareFontShape{T1}{pzc}{m}{sl}{<->ssub * pzc/mb/sl} {}
\DeclareFontShape{T1}{pzc}{m}{n}{<->ssub * pzc/mb/it} {}
\usepackage{chancery}
\usepackage{mathastext}
\linespread{1.05}""",
)
urwzccmg.add_to_document(
    r"""
\boldmath
""",
)
)

# Comic Sans MS
comicsansms = _new_ams_template()
comicsansms.description = "Comic Sans MS"
comicsansms.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Comic Sans MS}
\usepackage[defaultmathsizes]{mathastext}
""",
)
comicsansms.tex_compiler = "xelatex"
comicsansms.output_format = ".xdv"

# GFS Didot (Italic)
italicgfsdidot = _new_ams_template()
italicgfsdidot.description = "GFS Didot (Italic)"
italicgfsdidot.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\renewcommand\rmdefault{udidot}

```

```

\usepackage[LGRgreek, defaultmathsizes, italic]{mathastext}
\let\varphi\phi
"",
)

# Chalkduster
chalkduster = _new_ams_template()
chalkduster.description = "Chalkduster"
chalkduster.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Chalkduster}
\usepackage[defaultmathsizes]{mathastext}
""",
)
chalkduster.tex_compiler = "lualatex"
chalkduster.output_format = ".pdf"

# Minion Pro (and TX fonts symbols)
mptx = _new_ams_template()
mptx.description = "Minion Pro (and TX fonts symbols)"
mptx.add_to_preamble(
    r"""
\usepackage{txfonts}
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Minion Pro}
\usepackage[defaultmathsizes]{mathastext}
""",
)
mptx.tex_compiler = "xelatex"
mptx.output_format = ".xdv"

# GNU FreeSerif and FreeSans
gnufssfs = _new_ams_template()
gnufssfs.description = "GNU FreeSerif and FreeSans"
gnufssfs.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[ExternalLocation,
            Mapping=tex-text,
            BoldFont=FreeSerifBold,
            ItalicFont=FreeSerifItalic,
            BoldItalicFont=FreeSerifBoldItalic]{FreeSerif}
\setsansfont[ExternalLocation,
            Mapping=tex-text,
            BoldFont=FreeSansBold,
            ItalicFont=FreeSansOblique,
            BoldItalicFont=FreeSansBoldOblique,
            Scale=MatchLowercase]{FreeSans}
\renewcommand{\familydefault}{lmss}
\usepackage[LGRgreek, defaultmathsizes, noasterisk]{mathastext}
\renewcommand{\familydefault}{\sfdefault}
\Mathastext
\let\varphi\phi % no `var' phi in LGR encoding
\renewcommand{\familydefault}{\rmdefault}
""",
)
gnufssfs.tex_compiler = "xelatex"
gnufssfs.output_format = ".xdv"

# GFS NeoHellenic
gfsneohellenic = _new_ams_template()
gfsneohellenic.description = "GFS NeoHellenic"
gfsneohellenic.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\renewcommand{\rmdefault}{neohellenic}
\usepackage[LGRgreek]{mathastext}
\let\varphi\phi
\linespread{1.06}
""",
)
gnufssfs.tex_compiler = "xelatex"
gnufssfs.output_format = ".xdv"

# ECF Tall Paul (with Symbol font)
ecftallpaul = _new_ams_template()
ecftallpaul.description = "ECF Tall Paul (with Symbol font)"
ecftallpaul.add_to_preamble(
    r"""
\DeclareFontFamily{T1}{ftp} {}
\DeclareFontShape{T1}{ftp}{m}{n} {
    <->s*[1.4] fpmw8t
}{} % increase size by factor 1.4
\renewcommand\familydefault{ftp} % emerald package

```

```

\usepackage[symbol]{mathastext}
\let\infty\infty
",
)

# Droid Sans (Italic)
italicdroidsans = _new_ams_template()
italicdroidsans.description = "Droid Sans (Italic)"
italicdroidsans.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[default]{droidsans}
\usepackage[LGRgreek,defaultmathsizes,italic]{mathastext}
\let\varphi\phi
""",
)

# Baskerville (Italic)
italicbaskerville = _new_ams_template()
italicbaskerville.description = "Baskerville (Italic)"
italicbaskerville.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Baskerville}
\usepackage[defaultmathsizes,italic]{mathastext}
""",
)
italicbaskerville.tex_compiler = "xelatex"
italicbaskerville.output_format = ".xdv"

# ECF JD (with TX fonts)
ecfjdtx = _new_ams_template()
ecfjdtx.description = "ECF JD (with TX fonts)"
ecfjdtx.add_to_preamble(
    r"""
\usepackage{txfonts}
\usepackage[upright]{txgreeks}
\renewcommand\familydefault\{fjd\} % emerald package
\usepackage{mathastext}
""",
)
ecfjdtx.add_to_document(
    r"""\mathversion{bold}
""",
)
 

# Antykwa PÃ³, tawskiego (TX Fonts for Greek and math symbols)
aptxgm = _new_ams_template()
aptxgm.description = "Antykwa PÃ³, tawskiego (TX Fonts for Greek and math symbols)"
aptxgm.add_to_preamble(
    r"""
\usepackage[OT4,OT1]{fontenc}
\usepackage{txfonts}
\usepackage[upright]{txgreeks}
\usepackage{antpol}
\usepackage[defaultmathsizes,nolessnomore]{mathastext}
""",
)
 

# Papyrus
papyrus = _new_ams_template()
papyrus.description = "Papyrus"
papyrus.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Papyrus}
\usepackage[defaultmathsizes]{mathastext}
""",
)
papyrus.tex_compiler = "xelatex"
papyrus.output_format = ".xdv"

# GNU FreeSerif (and TX fonts symbols)
gnufstx = _new_ams_template()
gnufstx.description = "GNU FreeSerif (and TX fonts symbols)"
gnufstx.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\usepackage{txfonts} %\let\mathbb=\varmathbb
\setmainfont[ExternalLocation,
    Mapping=tex-text,

```

```

BoldFont=FreeSerifBold,
ItalicFont=FreeSerifItalic,
BoldItalicFont=FreeSerifBoldItalic]{FreeSerif}
\usepackage[defaultmathsizes]{mathastext}
"",
)
gnufstx.tex_compiler = "xelatex"
gnufstx.output_format = ".pdf"

# ECF Skeetch (CM Greek)
ecfscmg = _new_ams_template()
ecfscmg.description = "ECF Skeetch (CM Greek)"
ecfscmg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[T1]{fontenc}
\DeclareFontFamily{T1}{fsk} {}
\DeclareFontShape{T1}{fsk}{m}{n}{<->s*[1.315] fskmw8t} {}
\renewcommand\rmdefault{fsk}
\usepackage[noendash,defaultmathsizes,nohbar,defaultimath]{mathastext}
"",
)

# Latin Modern Typewriter Proportional (CM Greek) (Italic)
italiclmtpcm = _new_ams_template()
italiclmtpcm.description = "Latin Modern Typewriter Proportional (CM Greek) (Italic)"
italiclmtpcm.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[variablett,nomath]{lmodern}
\renewcommand{\familydefault}{\ttdefault}
\usepackage[frenchmath]{mathastext}
\linespread{1.08}
"",
)

# Baskervald ADF with Fourier
baskervaldadff = _new_ams_template()
baskervaldadff.description = "Baskervald ADF with Fourier"
baskervaldadff.add_to_preamble(
    r"""
\usepackage[upright]{fourier}
\usepackage{baskervald}
\usepackage[defaultmathsizes,noasterisk]{mathastext}
"",
)

# Droid Serif (PX math symbols) (Italic)
italicdroidserifpx = _new_ams_template()
italicdroidserifpx.description = "Droid Serif (PX math symbols) (Italic)"
italicdroidserifpx.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{pxfonts}
\usepackage[default]{droidserif}
\usepackage[LGRgreek,defaultmathsizes,italic,basic]{mathastext}
\let\varphi\phi
"",
)

# Biolinum
biolinum = _new_ams_template()
biolinum.description = "Biolinum"
biolinum.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{libertine}
\renewcommand{\familydefault}{\sfdefault}
\usepackage[greek=n,biolinum]{libgreek}
\usepackage[noasterisk,defaultmathsizes]{mathastext}
"",
)

# Vollkorn with Fourier (Italic)
italicvollkornf = _new_ams_template()
italicvollkornf.description = "Vollkorn with Fourier (Italic)"
italicvollkornf.add_to_preamble(
    r"""
\usepackage{fourier}
\usepackage{vollkorn}
\usepackage[italic,nohbar]{mathastext}

```

```

"""
)

# Chalkboard SE
chalkboardse = _new_ams_template()
chalkboardse.description = "Chalkboard SE"
chalkboardse.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Chalkboard SE}
\usepackage[defaultmathsizes]{mathastext}
""",
)
chalkboardse.tex_compiler = "xelatex"
chalkboardse.output_format = ".xdv"

# Noteworthy Light
noteworthylight = _new_ams_template()
noteworthylight.description = "Noteworthy Light"
noteworthylight.add_to_preamble(
    r"""
\usepackage[no-math]{fontspec}
\setmainfont[Mapping=tex-text]{Noteworthy Light}
\usepackage[defaultmathsizes]{mathastext}
""",
)
noteworthylight.tex_compiler = "xelatex"
noteworthylight.output_format = ".xdv"

# Epigrafica
epigrafica = _new_ams_template()
epigrafica.description = "Epigrafica"
epigrafica.add_to_preamble(
    r"""
\usepackage[LGR,OT1]{fontenc}
\usepackage{epigrafica}
\usepackage[basic,LGRgreek,defaultmathsizes]{mathastext}
\let\varphi\phi
\linespread{1.2}
""",
)
epigrafica.tex_compiler = "xelatex"
epigrafica.output_format = ".xdv"

# Libris ADF with Fourier
librisadff = _new_ams_template()
librisadff.description = "Libris ADF with Fourier"
librisadff.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[upright]{fourier}
\usepackage{libris}
\renewcommand{\familydefault}{\sfdefault}
\usepackage[noasterisk]{mathastext}
""",
)
librisadff.tex_compiler = "xelatex"
librisadff.output_format = ".xdv"

# Venturis ADF with Fourier (Italic)
italicvanturisadff = _new_ams_template()
italicvanturisadff.description = "Venturis ADF with Fourier (Italic)"
italicvanturisadff.add_to_preamble(
    r"""
\usepackage{fourier}
\usepackage[lf]{venturis}
\usepackage[italic,defaultmathsizes,noasterisk]{mathastext}
""",
)
italicvanturisadff.tex_compiler = "xelatex"
italicvanturisadff.output_format = ".xdv"

# GFS Bodoni
gfsbodoni = _new_ams_template()
gfsbodoni.description = "GFS Bodoni"
gfsbodoni.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\renewcommand{\rmdefault}{bodoni}
\usepackage[LGRgreek]{mathastext}
\let\varphi\phi
\linespread{1.06}
""",
)
gfsbodoni.tex_compiler = "xelatex"
gfsbodoni.output_format = ".xdv"

# BrushScriptX-Italic (PX math and Greek)
brushscriptpxp = _new_ams_template()
brushscriptpxp.description = "BrushScriptX-Italic (PX math and Greek)"
brushscriptpxp.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[OT1]{fontenc}
\usepackage{brushscriptx}
\usepackage[italic,defaultmathsizes,noasterisk]{mathastext}
\let\varphi\phi
\linespread{1.06}
""",
)
brushscriptpxp.tex_compiler = "xelatex"
brushscriptpxp.output_format = ".xdv"

```

```

brushscriptpx.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{pxfonts}
%\usepackage{pbsi}
\renewcommand{\rmdefault}{pbsi}
\renewcommand{\mddefault}{xl}
\renewcommand{\bfdefault}{xl}
\usepackage[defaultmathsizes,noasterisk]{mathastext}
""",
)
brushscriptpx.add_to_document(
    r"""\boldmath
""",
)
brushscriptpx.tex_compiler = "xelatex"
brushscriptpx.output_format = ".xdv"

# URW Avant Garde (Symbol Greek)
urwagsg = _new_ams_template()
urwagsg.description = "URW Avant Garde (Symbol Greek)"
urwagsg.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage{avant}
\renewcommand{\familydefault}{\sfdefault}
\usepackage[symbolgreek,defaultmathsizes]{mathastext}
""",
)
# Times with Fourier (Italic)
italictimesf = _new_ams_template()
italictimesf.description = "Times with Fourier (Italic)"
italictimesf.add_to_preamble(
    r"""
\usepackage{fourier}
\renewcommand{\rmdefault}{ptm}
\usepackage[italic,defaultmathsizes,noasterisk]{mathastext}
""",
)
# Helvetica with Fourier (Italic)
italichelveticaaf = _new_ams_template()
italichelveticaaf.description = "Helvetica with Fourier (Italic)"
italichelveticaaf.add_to_preamble(
    r"""
\usepackage[T1]{fontenc}
\usepackage[scaled]{helvet}
\usepackage{fourier}
\renewcommand{\rmdefault}{phv}
\usepackage[italic,defaultmathsizes,noasterisk]{mathastext}
""",
)

class TexFontTemplates:
    """
    A collection of TeX templates for the fonts described at http://jf.burnol.free.fr/showcase.html

    These templates are specifically designed to allow you to typeset formulae and mathematics using different fonts. They are based on the mathastext LaTeX package.

    Examples
    -----
    Normal usage as a value for the keyword argument tex_template of Tex() and MathTex() mobjects::

        ``Tex("My TeX code", tex_template=TexFontTemplates.comic_sans)``
    Notes
    -----
    Many of these templates require that specific fonts are installed on your local machine.
    For example, choosing the template TexFontTemplates.comic_sans will not compile if the Comic Sans Microsoft font is not installed.

    To experiment, try to render the TexFontTemplateLibrary example scene:
        ``manim path/to/manim/example_scenes/advanced_tex_fonts.py TexFontTemplateLibrary -p -ql``
    """

    american_typewriter = americantypewriter
    """American Typewriter"""
    antykwa = aptxgm
    """Antykwa PÅ'Ä, tawskiego (TX Fonts for Greek and math symbols)"""
    apple_chancery = applechancery

```

""""Apple Chancery"""
auriocus_kalligraphicus = aksq
"""Auriocus Kalligraphicus (Symbol Greek)"""
baskervald_adf_fourier = baskervaldadff
"""Baskervald ADF with Fourier"""
baskerville_it = italicbaskerville
"""Baskerville (Italic)"""
biolinum = biolinum
"""Biolinum"""
brushscriptx = brushscriptpx
"""BrushScriptX-Italic (PX math and Greek)"""
chalkboard_se = chalkboardse
"""Chalkboard SE"""
chalkduster = chalkduster
"""Chalkduster"""
comfortaa = comfortaa
"""Comfortaa"""
comic_sans = comicsansms
"""Comic Sans MS"""
droid_sans = droidsans
"""Droid Sans"""
droid_sans_it = italicdroidsans
"""Droid Sans (Italic)"""
droid_serif = droidserif
"""Droid Serif"""
droid_serif_px_it = italicdroidserifpx
"""Droid Serif (PX math symbols) (Italic)"""
ecf_augie = ecfaugeeg
"""ECF Augie (Euler Greek)"""
ecf_jd = ecfjdtx
"""ECF JD (with TX fonts)"""
ecf_skeetch = ecfscmg
"""ECF Skeetch (CM Greek)"""
ecf_tall_paul = ecftallpaul
"""ECF Tall Paul (with Symbol font)"""
ecf_webster = ecfwebstertx
"""ECF Webster (with TX fonts)"""
electrum_adf = electrumadfcn
"""Electrum ADF (CM Greek)"""
epigrafica = epigrafica
""" Epigrafica """
fourier_utopia = fufug
"""Fourier Utopia (Fourier upright Greek)"""
french_cursive = fcsg
"""French Cursive (Euler Greek)"""
gfs_bodoni = gfsbodoni
"""GFS Bodoni"""
gfs_didot = italicgfsdidot
"""GFS Didot (Italic)"""
gfs_neoHellenic = gfsneohellenic
"""GFS NeoHellenic"""
gnu_freesans_tx = gnufstx
"""GNU FreeSerif (and TX fonts symbols)"""
gnu_freeserif_freesans = gnufsf
"""GNU FreeSerif and FreeSans"""
helvetica_fourier_it = italichelvetica
"""Helvetica with Fourier (Italic)"""
latin_modern_tw_it = italiclmtpcm
"""Latin Modern Typewriter Proportional (CM Greek) (Italic)"""
latin_modern_tw = lmtp
"""Latin Modern Typewriter Proportional"""
libertine = libertine
"""Libertine"""
libris_adf_fourier = librisadff
"""Libris ADF with Fourier"""
minion_pro_myriad_pro = mpmpx
"""Minion Pro and Myriad Pro (and TX fonts symbols)"""
minion_pro_tx = mpx
"""Minion Pro (and TX fonts symbols)"""
new_century_schoolbook = ncsg
"""New Century Schoolbook (Symbol Greek)"""
new_century_schoolbook_px = ncsgpxm
"""New Century Schoolbook (Symbol Greek, PX math symbols)"""
noteworthy_light = noteworthylight
"""Noteworthy Light"""
palatino = palatinosg
"""Palatino (Symbol Greek)"""
papyrus = papyrus
"""Papyrus"""
romande_adf_fourier_it = italicromandeadff
"""Romande ADF with Fourier (Italic)"""
slitex = slitexg
"""SliTeX (Euler Greek)"""
times_fourier_it = italictimesf
"""Times with Fourier (Italic)"""
urw_avant_garde = urwags
"""URW Avant Garde (Symbol Greek)"""

```
urw_zapf_chancery = urwzccmgs
"""URW Zapf Chancery (CM Greek)"""
venturis_adf_fourier_it = italicvanturisadff
"""Venturis ADF with Fourier (Italic)"""
verdana_it = italicverdana
"""Verdana (Italic)"""
vollkorn_fourier_it = italicvollkornf
"""Vollkorn with Fourier (Italic)"""
vollkorn = vollkorntx
"""Vollkorn (TX fonts for Greek and math symbols)"""
zapf_chancery = zapfchancery
"""Zapf Chancery"""
```

transform_matching_parts.py

```
"""Animations that try to transform Mobjects while keeping track of identical parts."""

from __future__ import annotations

__all__ = ["TransformMatchingShapes", "TransformMatchingTex"]

from typing import TYPE_CHECKING

import numpy as np

from manim.mobject.opengl.opengl_mobject import OpenGLGroup, OpenGLMobject
from manim.mobject.opengl_vectorized_mobject import OpenGLVGroup, OpenGLVGroup

from ..config import config
from ..constants import RendererType
from ..mobject.mobject import Group, Mobject
from ..mobject.types.vectorized_mobject import VGroup, VMobject
from .composition import AnimationGroup
from .fading import FadeIn, FadeOut
from .transform import FadeTransformPieces, Transform

if TYPE_CHECKING:
    from ..scene.scene import Scene

class TransformMatchingAbstractBase(AnimationGroup):
    """Abstract base class for transformations that keep track of matching parts.

    Subclasses have to implement the two static methods
    :meth:`~.TransformMatchingAbstractBase.get_mobject_parts` and
    :meth:`~.TransformMatchingAbstractBase.get_mobject_key`.

    Basically, this transformation first maps all submobjects returned
    by the ``get_mobject_parts`` method to certain keys by applying the
    ``get_mobject_key`` method. Then, submobjects with matching keys
    are transformed into each other.

    Parameters
    -----
    mobject
        The starting :class:`~.Mobject`.
    target_mobject
        The target :class:`~.Mobject`.
    transform_mismatches
        Controls whether submobjects without a matching key are transformed
        into each other by using :class:`~.Transform`. Default: ``False``.
    fade_transform_mismatches
        Controls whether submobjects without a matching key are transformed
        into each other by using :class:`~.FadeTransform`. Default: ``False``.
    key_map
        Optional. A dictionary mapping keys belonging to some of the starting mobject's
        submobjects (i.e., the return values of the ``get_mobject_key`` method)
        to some keys belonging to the target mobject's submobjects that should
        be transformed although the keys don't match.
    kwargs
        All further keyword arguments are passed to the submobject transformations.

    Note
    -----
    If neither ``transform_mismatches`` nor ``fade_transform_mismatches``
    are set to ``True``, submobjects without matching keys in the starting
    mobject are faded out in the direction of the unmatched submobjects in
    the target mobject, and unmatched submobjects in the target mobject
    are faded in from the direction of the unmatched submobjects in the
    start mobject.

    """

    def __init__(
        self,
        mobject: Mobject,
        target_mobject: Mobject,
        transform_mismatches: bool = False,
        fade_transform_mismatches: bool = False,
        key_map: dict | None = None,
        **kwargs,
    ):
        if isinstance(mobject, OpenGLVGroup):
            group_type = OpenGLVGroup
        elif isinstance(mobject, OpenGLMobject):
            group_type = OpenGLMobject
```

```

        group_type = OpenGLGroup
    elif isinstance(mobject, VMobject):
        group_type = VGroup
    else:
        group_type = Group

    source_map = self.get_shape_map(mobject)
    target_map = self.get_shape_map(target_mobject)

    if key_map is None:
        key_map = {}

    # Create two mobjects whose submobjects all match each other
    # according to whatever keys are used for source_map and
    # target_map
    transform_source = group_type()
    transform_target = group_type()
    kwargs["final_alpha_value"] = 0
    for key in set(source_map).intersection(target_map):
        transform_source.add(source_map[key])
        transform_target.add(target_map[key])
    anims = [Transform(transform_source, transform_target, **kwargs)]
    # User can manually specify when one part should transform
    # into another despite not matching by using key_map
    key_mapped_source = group_type()
    key_mapped_target = group_type()
    for key1, key2 in key_map.items():
        if key1 in source_map and key2 in target_map:
            key_mapped_source.add(source_map[key1])
            key_mapped_target.add(target_map[key2])
            source_map.pop(key1, None)
            target_map.pop(key2, None)
    if len(key_mapped_source) > 0:
        anims.append(
            FadeTransformPieces(key_mapped_source, key_mapped_target, **kwargs),
        )

    fade_source = group_type()
    fade_target = group_type()
    for key in set(source_map).difference(target_map):
        fade_source.add(source_map[key])
    for key in set(target_map).difference(source_map):
        fade_target.add(target_map[key])
    fade_target_copy = fade_target.copy()

    if transform_mismatches:
        if "replace_mobject_with_target_in_scene" not in kwargs:
            kwargs["replace_mobject_with_target_in_scene"] = True
        anims.append(Transform(fade_source, fade_target, **kwargs))
    elif fade_transform_mismatches:
        anims.append(FadeTransformPieces(fade_source, fade_target, **kwargs))
    else:
        anims.append(FadeOut(fade_source, target_position=fade_target, **kwargs))
        anims.append(
            FadeIn(fade_target_copy, target_position=fade_target, **kwargs),
        )

    super().__init__(*anims)

    self.to_remove = [mobject, fade_target_copy]
    self.to_add = target_mobject

def get_shape_map(self, mobject: Mobject) -> dict:
    shape_map = {}
    for sm in self.get_mobject_parts(mobject):
        key = self.get_mobject_key(sm)
        if key not in shape_map:
            if config["renderer"] == RendererType.OPENGL:
                shape_map[key] = OpenGLGroup()
            else:
                shape_map[key] = VGroup()
        shape_map[key].add(sm)
    return shape_map

def clean_up_from_scene(self, scene: Scene) -> None:
    # Interpolate all animations back to 0 to ensure source mobjects remain unchanged.
    for anim in self.animations:
        anim.interpolate(0)
    scene.remove(self.mobject)
    scene.remove(*self.to_remove)
    scene.add(self.to_add)

@staticmethod
def get_mobject_parts(mobject: Mobject):
    raise NotImplementedError("To be implemented in subclass.")

@staticmethod

```

```

def get_mobject_key(mobject: Mobject):
    raise NotImplementedError("To be implemented in subclass.")


class TransformMatchingShapes(TransformMatchingAbstractBase):
    """An animation trying to transform groups by matching the shape
    of their subobjects.

    Two subobjects match if the hash of their point coordinates after
    normalization (i.e., after translation to the origin, fixing the submobject
    height at 1 unit, and rounding the coordinates to three decimal places)
    matches.

    See also
    -----
    :class:`~.TransformMatchingAbstractBase`


    Examples
    -----
    .. manim:: Anagram

        class Anagram(Scene):
            def construct(self):
                src = Text("the morse code")
                tar = Text("here come dots")
                self.play(Write(src))
                self.wait(0.5)
                self.play(TransformMatchingShapes(src, tar, path_arc=PI/2))
                self.wait(0.5)

    """

    def __init__(
        self,
        mobject: Mobject,
        target_mobject: Mobject,
        transform_mismatches: bool = False,
        fade_transform_mismatches: bool = False,
        key_map: dict | None = None,
        **kwargs,
    ):
        super().__init__(
            mobject,
            target_mobject,
            transform_mismatches=transform_mismatches,
            fade_transform_mismatches=fade_transform_mismatches,
            key_map=key_map,
            **kwargs,
        )

    @staticmethod
    def get_mobject_parts(mobject: Mobject) -> list[Mobject]:
        return mobject.family_members_with_points()

    @staticmethod
    def get_mobject_key(mobject: Mobject) -> int:
        mobject.save_state()
        mobject.center()
        mobject.set_height(1)
        result = hash(np.round(mobject.points, 3).tobytes())
        mobject.restore()
        return result


class TransformMatchingTex(TransformMatchingAbstractBase):
    """A transformation trying to transform rendered LaTeX strings.

    Two subobjects match if their ``tex_string`` matches.

    See also
    -----
    :class:`~.TransformMatchingAbstractBase`


    Examples
    -----
    .. manim:: MatchingEquationParts

        class MatchingEquationParts(Scene):
            def construct(self):
                variables = VGroup(MathTex("a"), MathTex("b"), MathTex("c")).arrange_submobjects().shift(UP)
                eq1 = MathTex("{x}^2", "+", "{y}^2", "=", "{z}^2")
                eq2 = MathTex("{a}^2", "+", "{b}^2", "=", "{c}^2")
                eq3 = MathTex("{a}^2", "=", "{c}^2", "-", "{b}^2")

```

```

        self.add(eq1)
        self.wait(0.5)
        self.play(TransformMatchingTex(Group(eq1, variables), eq2))
        self.wait(0.5)
        self.play(TransformMatchingTex(eq2, eq3))
        self.wait(0.5)

"""

def __init__(
    self,
    mobject: Mobject,
    target_mobject: Mobject,
    transform_mismatches: bool = False,
    fade_transform_mismatches: bool = False,
    key_map: dict | None = None,
    **kwargs,
):
    super().__init__(
        mobject,
        target_mobject,
        transform_mismatches=transform_mismatches,
        fade_transform_mismatches=fade_transform_mismatches,
        key_map=key_map,
        **kwargs,
    )

@staticmethod
def get_mobject_parts(mobject: Mobject) -> list[Mobject]:
    if isinstance(mobject, (Group, VGroup, OpenGLGroup, OpenGLVGGroup)):
        return [
            p
            for s in mobject.submobjects
            for p in TransformMatchingTex.get_mobject_parts(s)
        ]
    else:
        assert hasattr(mobject, "tex_string")
        return mobject.submobjects

@staticmethod
def get_mobject_key(mobject: Mobject) -> str:
    return mobject.tex_string

```

multi_camera.py

```
"""A camera supporting multiple perspectives."""

from __future__ import annotations

__all__ = ["MultiCamera"]

from manim.mobject.types.image_mobject import ImageMobject
from ..camera.moving_camera import MovingCamera
from ..utils.iterables import list_difference_update

class MultiCamera(MovingCamera):
    """Camera Object that allows for multiple perspectives."""

    def __init__(self,
                 image_mobjects_from_cameras: ImageMobject | None = None,
                 allow_cameras_to_capture_their_own_display=False,
                 **kwargs,
                 ):
        """Initialises the MultiCamera

        Parameters
        -----
        image_mobjects_from_cameras

        kwargs
            Any valid keyword arguments of MovingCamera.

        """
        self.image_mobjects_from_cameras = []
        if image_mobjects_from_cameras is not None:
            for imfc in image_mobjects_from_cameras:
                self.add_image_mobject_from_camera(imfc)
        self.allow_cameras_to_capture_their_own_display = (
            allow_cameras_to_capture_their_own_display
        )
        super().__init__(**kwargs)

    def add_image_mobject_from_camera(self, image_mobject_from_camera: ImageMobject):
        """Adds an ImageMobject that's been obtained from the camera
        into the list ``self.image_mobject_from_cameras``

        Parameters
        -----
        image_mobject_from_camera
            The ImageMobject to add to self.image_mobject_from_cameras

        """
        # A silly method to have right now, but maybe there are things
        # we want to guarantee about any imfc's added later.
        imfc = image_mobject_from_camera
        assert isinstance(imfc.camera, MovingCamera)
        self.image_mobjects_from_cameras.append(imfc)

    def update_sub_cameras(self):
        """Reshape sub_camera pixel arrays"""
        for imfc in self.image_mobjects_from_cameras:
            pixel_height, pixel_width = self.pixel_array.shape[:2]
            imfc.camera.frame_shape = (
                imfc.camera.frame.height,
                imfc.camera.frame.width,
            )
            imfc.camera.reset_pixel_shape(
                int(pixel_height * imfc.height / self.frame_height),
                int(pixel_width * imfc.width / self.frame_width),
            )

    def reset(self):
        """Resets the MultiCamera.

        Returns
        -----
        MultiCamera
            The reset MultiCamera

        """
        for imfc in self.image_mobjects_from_cameras:
            imfc.camera.reset()
        super().reset()
        return self
```

```
def capture_mobjects(self, mobjects, **kwargs):
    self.update_sub_cameras()
    for imfc in self.image_mobjects_from_cameras:
        to_add = list(mobjects)
        if not self.allow_cameras_to_capture_their_own_display:
            to_add = list_difference_update(to_add, imfc.get_family())
        imfc.camera.capture_mobjects(to_add, **kwargs)
    super().capture_mobjects(mobjects, **kwargs)

def get_mobjects_indicating_movement(self):
    """Returns all mobjects whose movement implies that the camera
    should think of all other mobjects on the screen as moving

    Returns
    ------
    list
    """
    return [self.frame] + [
        imfc.camera.frame for imfc in self.image_mobjects_from_cameras
    ]
```

bezier.py

```
"""Utility functions related to Bézier curves."""

from __future__ import annotations

__all__ = [
    "bezier",
    "partial_bezier_points",
    "partial_quadratic_bezier_points",
    "interpolate",
    "integer_interpolate",
    "mid",
    "inverse_interpolate",
    "match_interpolate",
    "get_smooth_handle_points",
    "get_smooth_cubic_bezier_handle_points",
    "diag_to_matrix",
    "is_closed",
    "proportions_along_bezier_curve_for_point",
    "point_lies_on_bezier",
]
]

import typing
from functools import reduce
from typing import Iterable

import numpy as np
from scipy import linalg

from ..utils.simple_functions import choose
from ..utils.space_ops import cross2d, find_intersection

def bezier(
    points: np.ndarray,
) -> typing.Callable[[float], int | typing.Iterable]:
    """Classic implementation of a bezier curve.

    Parameters
    -----
    points
        points defining the desired bezier curve.

    Returns
    -----
    typing.Callable[[float], typing.Union[int, typing.Iterable]]
        function describing the bezier curve.
    """
    n = len(points) - 1

    # Cubic Bezier curve
    if n == 3:
        return (
            lambda t: (1 - t) ** 3 * points[0]
            + 3 * t * (1 - t) ** 2 * points[1]
            + 3 * (1 - t) * t**2 * points[2]
            + t**3 * points[3]
        )
    # Quadratic Bezier curve
    if n == 2:
        return (
            lambda t: (1 - t) ** 2 * points[0]
            + 2 * t * (1 - t) * points[1]
            + t**2 * points[2]
        )

    return lambda t: sum(
        ((1 - t) ** (n - k)) * (t**k) * choose(n, k) * point
        for k, point in enumerate(points)
    )

def partial_bezier_points(points: np.ndarray, a: float, b: float) -> np.ndarray:
    """Given an array of points which define bezier curve, and two numbers 0<=a<=b<=1, return an array of the same size, which describes the portion of the original bezier curve on the interval [a, b]."""

    This algorithm is pretty nifty, and pretty dense.

    Parameters
    -----
    points
        set of points defining the bezier curve.
    a
        lower bound of the desired partial bezier curve.
    b
        upper bound of the desired partial bezier curve.
```

```

upper bound of the desired partial bezier curve.

>Returns
-----
np.ndarray
    Set of points defining the partial bezier curve.
"""
if a == 1:
    return [points[-1]] * len(points)

a_to_1 = np.array([bezier(points[i:])(a) for i in range(len(points))])
end_prop = (b - a) / (1.0 - a)
return np.array([bezier(a_to_1[: i + 1])(end_prop) for i in range(len(points))])

# Shortened version of partial_bezier_points just for quadratics,
# since this is called a fair amount
def partial_quadratic_bezier_points(points, a, b):
    points = np.asarray(points, dtype=np.float64)
    if a == 1:
        return 3 * [points[-1]]

    def curve(t):
        return (
            points[0] * (1 - t) * (1 - t)
            + 2 * points[1] * t * (1 - t)
            + points[2] * t * t
        )

    # bezier(points)
    h0 = curve(a) if a > 0 else points[0]
    h2 = curve(b) if b < 1 else points[2]
    h1_prime = (1 - a) * points[1] + a * points[2]
    end_prop = (b - a) / (1.0 - a)
    h1 = (1 - end_prop) * h0 + end_prop * h1_prime
    return [h0, h1, h2]

def split_quadratic_bezier(points: np.ndarray, t: float) -> np.ndarray:
    """Split a quadratic Bézier curve at argument ``t`` into two quadratic curves.

>Parameters
-----
points
    The control points of the bezier curve
    has shape ``[a1, h1, b1]``

t
    The ``t``-value at which to split the Bézier curve

>Returns
-----
The two Bézier curves as a list of tuples,
has the shape ``[a1, h1, b1], [a2, h2, b2]``

"""
a1, h1, a2 = points
s1 = interpolate(a1, h1, t)
s2 = interpolate(h1, a2, t)
p = interpolate(s1, s2, t)

return np.array([a1, s1, p, p, s2, a2])

def subdivide_quadratic_bezier(points: Iterable[float], n: int) -> np.ndarray:
    """Subdivide a quadratic Bézier curve into ``n`` subcurves which have the same shape.

>The points at which the curve is split are located at the
arguments :math:`t = i/n` for :math:`i = 1, \dots, n-1`.

>Parameters
-----
points
    The control points of the Bézier curve in form ``[a1, h1, b1]``

n
    The number of curves to subdivide the Bézier curve into

>Returns
-----
The new points for the Bézier curve in the form ``[a1, h1, b1, a2, h2, b2, ...]``

.. image:: /_static/bezier_subdivision_example.png

"""
beziers = np.empty((n, 3, 3))
current = points
for j in range(0, n):
    i = n - j
    tmp = split_quadratic_bezier(current, 1 / i)
    beziers[j] = tmp[:3]
    current = tmp[3:]

```

```

return beziers.reshape(-1, 3)

def quadratic_bezier_remap(
    triplets: Iterable[Iterable[float]], new_number_of_curves: int
):
    """Remaps the number of curves to a higher amount by splitting bezier curves

Parameters
-----
triplets
    The triplets of the quadratic bezier curves to be remapped shape(n, 3, 3)

new_number_of_curves
    The number of curves that the output will contain. This needs to be higher than the current number.

Returns
-----
The new triplets for the quadratic bezier curves.

"""
difference = new_number_of_curves - len(triplets)
if difference <= 0:
    return triplets
new_triplets = np.zeros((new_number_of_curves, 3, 3))
idx = 0
for triplet in triplets:
    if difference > 0:
        tmp_noc = int(np.ceil(difference / len(triplets))) + 1
        tmp = subdivide_quadratic_bezier(triplet, tmp_noc).reshape(-1, 3, 3)
        for i in range(tmp_noc):
            new_triplets[idx + i] = tmp[i]
        difference -= tmp_noc - 1
        idx += tmp_noc
    else:
        new_triplets[idx] = triplet
        idx += 1
return new_triplets

"""
This is an alternate version of the function just for documentation purposes
-----

difference = new_number_of_curves - len(triplets)
if difference <= 0:
    return triplets
new_triplets = []
for triplet in triplets:
    if difference > 0:
        tmp_noc = int(np.ceil(difference / len(triplets))) + 1
        tmp = subdivide_quadratic_bezier(triplet, tmp_noc).reshape(-1, 3, 3)
        for i in range(tmp_noc):
            new_triplets.append(tmp[i])
        difference -= tmp_noc - 1
    else:
        new_triplets.append(triplet)
return new_triplets
"""

# Linear interpolation variants
def interpolate(start: np.ndarray, end: np.ndarray, alpha: float) -> np.ndarray:
    return (1 - alpha) * start + alpha * end

def integer_interpolate(
    start: float,
    end: float,
    alpha: float,
) -> tuple[int, float]:
    """
Alpha is a float between 0 and 1. This returns
an integer between start and end (inclusive) representing
appropriate interpolation between them, along with a
"residue" representing a new proportion between the
returned integer and the next one of the
list.

For example, if start=0, end=10, alpha=0.46, This
would return (4, 0.6).
"""
    if alpha >= 1:
        return (end - 1, 1.0)
    if alpha <= 0:
        return (start, 0)
    value = int(interpolate(start, end, alpha))
    residue = ((end - start) * alpha) % 1
    return (value, residue)

def mid(start: float, end: float) -> float:
    return (start + end) / 2.0

```

```

def inverse_interpolate(start: float, end: float, value: float) -> np.ndarray:
    return np.true_divide(value - start, end - start)

def match_interpolate(
    new_start: float,
    new_end: float,
    old_start: float,
    old_end: float,
    old_value: float,
) -> np.ndarray:
    return interpolate(
        new_start,
        new_end,
        inverse_interpolate(old_start, old_end, old_value),
    )

# Figuring out which bezier curves most smoothly connect a sequence of points

def get_smooth_cubic_bezier_handle_points(points):
    points = np.array(points)
    num_handles = len(points) - 1
    dim = points.shape[1]
    if num_handles < 1:
        return np.zeros((0, dim)), np.zeros((0, dim))
    # Must solve 2*num_handles equations to get the handles.
    # l and u are the number of lower an upper diagonal rows
    # in the matrix to solve.
    l, u = 2, 1
    # diag is a representation of the matrix in diagonal form
    # See https://www.particleincell.com/2012/bezier-splines/
    # for how to arrive at these equations
    diag = np.zeros((l + u + 1, 2 * num_handles))
    diag[0, 1::2] = -1
    diag[0, 2::2] = 1
    diag[1, 0::2] = 2
    diag[1, 1::2] = 1
    diag[2, 1:-2:2] = -2
    diag[3, 0:-3:2] = 1
    # last
    diag[2, -2] = -1
    diag[1, -1] = 2
    # This is the b as in Ax = b, where we are solving for x,
    # and A is represented using diag. However, think of entries
    # to x and b as being points in space, not numbers
    b = np.zeros((2 * num_handles, dim))
    b[1::2] = 2 * points[1:]
    b[0] = points[0]
    b[-1] = points[-1]

    def solve_func(b):
        return linalg.solve_banded((l, u), diag, b)

    use_closed_solve_function = is_closed(points)
    if use_closed_solve_function:
        # Get equations to relate first and last points
        matrix = diag_to_matrix((l, u), diag)
        # last row handles second derivative
        matrix[-1, [0, 1, -2, -1]] = [2, -1, 1, -2]
        # first row handles first derivative
        matrix[0, :] = np.zeros(matrix.shape[1])
        matrix[0, [0, -1]] = [1, 1]
        b[0] = 2 * points[0]
        b[-1] = np.zeros(dim)

        def closed_curve_solve_func(b):
            return linalg.solve(matrix, b)

        handle_pairs = np.zeros((2 * num_handles, dim))
        for i in range(dim):
            if use_closed_solve_function:
                handle_pairs[:, i] = closed_curve_solve_func(b[:, i])
            else:
                handle_pairs[:, i] = solve_func(b[:, i])
        return handle_pairs[0::2], handle_pairs[1::2]

    def get_smooth_handle_points(
        points: np.ndarray,
    ) -> tuple[np.ndarray, np.ndarray]:
        """Given some anchors (points), compute handles so the resulting bezier curve is smooth.

        Parameters
        -----
        points
            Anchors.

```

```

>Returns
-----
typing.Tuple[np.ndarray, np.ndarray]
    Computed handles.
"""

# NOTE points here are anchors.
points = np.array(points)
num_handles = len(points) - 1
dim = points.shape[1]
if num_handles < 1:
    return np.zeros((0, dim)), np.zeros((0, dim))
# Must solve 2*num_handles equations to get the handles.
# l and u are the number of lower an upper diagonal rows
# in the matrix to solve.
l, u = 2, 1
# diag is a representation of the matrix in diagonal form
# See https://www.particleincell.com/2012/bezier-splines/
# for how to arrive at these equations
diag = np.zeros((l + u + 1, 2 * num_handles))
diag[0, 1::2] = -1
diag[0, 2::2] = 1
diag[1, 0::2] = 2
diag[1, 1::2] = 1
diag[2, 1:-2:2] = -2
diag[3, 0:-3:2] = 1
# last
diag[2, -2] = -1
diag[1, -1] = 2
# This is the b as in Ax = b, where we are solving for x,
# and A is represented using diag. However, think of entries
# to x and b as being points in space, not numbers
b = np.zeros((2 * num_handles, dim))
b[1::2] = 2 * points[1:]
b[0] = points[0]
b[-1] = points[-1]

def solve_func(b: np.ndarray) -> np.ndarray:
    return linalg.solve_banded((l, u), diag, b)

use_closed_solve_function = is_closed(points)
if use_closed_solve_function:
    # Get equations to relate first and last points
    matrix = diag_to_matrix((l, u), diag)
    # last row handles second derivative
    matrix[-1, [0, 1, -2, -1]] = [2, -1, 1, -2]
    # first row handles first derivative
    matrix[0, :] = np.zeros(matrix.shape[1])
    matrix[0, [0, -1]] = [1, 1]
    b[0] = 2 * points[0]
    b[-1] = np.zeros(dim)

    def closed_curve_solve_func(b: np.ndarray) -> np.ndarray:
        return linalg.solve(matrix, b)

handle_pairs = np.zeros((2 * num_handles, dim))
for i in range(dim):
    if use_closed_solve_function:
        handle_pairs[:, i] = closed_curve_solve_func(b[:, i])
    else:
        handle_pairs[:, i] = solve_func(b[:, i])
return handle_pairs[0::2], handle_pairs[1::2]

def diag_to_matrix(l_and_u: tuple[int, int], diag: np.ndarray) -> np.ndarray:
"""
Converts array whose rows represent diagonal
entries of a matrix into the matrix itself.
See scipy.linalg.solve_banded
"""

l, u = l_and_u
dim = diag.shape[1]
matrix = np.zeros((dim, dim))
for i in range(l + u + 1):
    np.fill_diagonal(
        matrix[max(0, i - u) :, max(0, u - i) :],
        diag[i, max(0, u - i) :],
    )
return matrix

# Given 4 control points for a cubic bezier curve (or arrays of such)
# return control points for 2 quadratics (or 2n quadratics) approximating them.
def get_quadratic_approximation_of_cubic(a0, h0, h1, a1):
    a0 = np.array(a0, ndmin=2)
    h0 = np.array(h0, ndmin=2)
    h1 = np.array(h1, ndmin=2)
    a1 = np.array(a1, ndmin=2)
    # Tangent vectors at the start and end.
    T0 = h0 - a0
    T1 = a1 - h1

```

```

# Search for inflection points. If none are found, use the
# midpoint as a cut point.
# Based on http://www.caffeineowl.com/graphics/2d/vectorial/cubic-inflexion.html
has_infl = np.ones(len(a0), dtype=bool)

p = h0 - a0
q = h1 - 2 * h0 + a0
r = a1 - 3 * h1 + 3 * h0 - a0

a = cross2d(q, r)
b = cross2d(p, r)
c = cross2d(p, q)

disc = b * b - 4 * a * c
has_infl &= disc > 0
sqrt_disc = np.sqrt(np.abs(disc))
settings = np.seterr(all="ignore")
ti_bounds = []
for sgn in [-1, +1]:
    ti = (-b + sgn * sqrt_disc) / (2 * a)
    ti[a == 0] = (-c / b)[a == 0]
    ti[(a == 0) & (b == 0)] = 0
    ti_bounds.append(ti)
ti_min, ti_max = ti_bounds
np.seterr(**settings)
ti_min_in_range = has_infl & (0 < ti_min) & (ti_min < 1)
ti_max_in_range = has_infl & (0 < ti_max) & (ti_max < 1)

# Choose a value of t which starts at 0.5,
# but is updated to one of the inflection points
# if they lie between 0 and 1

t_mid = 0.5 * np.ones(len(a0))
t_mid[ti_min_in_range] = ti_min[ti_min_in_range]
t_mid[ti_max_in_range] = ti_max[ti_max_in_range]

m, n = a0.shape
t_mid = t_mid.repeat(n).reshape((m, n))

# Compute bezier point and tangent at the chosen value of t
mid = bezier([a0, h0, h1, a1])(t_mid)
Tm = bezier([h0 - a0, h1 - h0, a1 - h1])(t_mid)

# Intersection between tangent lines at end points
# and tangent in the middle
i0 = find_intersection(a0, T0, mid, Tm)
i1 = find_intersection(a1, T1, mid, Tm)

m, n = np.shape(a0)
result = np.zeros((6 * m, n))
result[0::6] = a0
result[1::6] = i0
result[2::6] = mid
result[3::6] = mid
result[4::6] = i1
result[5::6] = a1
return result

def is_closed(points: tuple[np.ndarray, np.ndarray]) -> bool:
    return np.allclose(points[0], points[-1])

def proportions_along_bezier_curve_for_point(
    point: typing.Iterable[float | int],
    control_points: typing.Iterable[typing.Iterable[float | int]],
    round_to: float | int | None = 1e-6,
) -> np.ndarray:
    """Obtains the proportion along the bezier curve corresponding to a given point
    given the bezier curve's control points.

    The bezier polynomial is constructed using the coordinates of the given point
    as well as the bezier curve's control points. On solving the polynomial for each dimension,
    if there are roots common to every dimension, those roots give the proportion along the
    curve the point is at. If there are no real roots, the point does not lie on the curve.

    Parameters
    -----
    point
        The Cartesian Coordinates of the point whose parameter
        should be obtained.
    control_points
        The Cartesian Coordinates of the ordered control
        points of the bezier curve on which the point may
        or may not lie.
    round_to
        A float whose number of decimal places all values
        such as coordinates of points will be rounded.

    Returns
    -----
    """

```

```

np.ndarray[float]
List containing possible parameters (the proportions along the bezier curve)
for the given point on the given bezier curve.
This usually only contains one or zero elements, but if the
point is, say, at the beginning/end of a closed loop, may return
a list with more than 1 value, corresponding to the beginning and
end etc. of the loop.

Raises
-----
:raises:`ValueError`
    When ``point`` and the control points have different shapes.
"""

# Method taken from
# http://polymathprogrammer.com/2012/04/03/does-point-lie-on-bezier-curve/

if not all(np.shape(point) == np.shape(c_p) for c_p in control_points):
    raise ValueError(
        f"Point {point} and Control Points {control_points} have different shapes.",
    )

control_points = np.array(control_points)
n = len(control_points) - 1

roots = []
for dim, coord in enumerate(point):
    control_coords = control_points[:, dim]
    terms = []
    for term_power in range(n, -1, -1):
        outercoeff = choose(n, term_power)
        term = []
        sign = 1
        for subterm_num in range(term_power, -1, -1):
            innercoeff = choose(term_power, subterm_num) * sign
            subterm = innercoeff * control_coords[subterm_num]
            if term_power == 0:
                subterm -= coord
            term.append(subterm)
            sign *= -1
        terms.append(outercoeff * sum(np.array(term)))
    if all(term == 0 for term in terms):
        # Then both Bezier curve and Point lie on the same plane.
        # Roots will be none, but in this specific instance, we don't need to consider that.
        continue
    bezier_poly = np.polynomial.Polynomial(terms[::-1])
    polynom_roots = bezier_poly.roots()
    if len(pynom_roots) > 0:
        polynom_roots = np.around(pynom_roots, int(np.log10(1 / round_to)))
    roots.append(pynom_roots)

roots = [root for root in rootlist if root.imag == 0] for rootlist in roots]
roots = reduce(np.intersect1d, roots) # Get common roots.
roots = np.array([r.real for r in roots if 0 <= r.real <= 1])
return roots

def point_lies_on_bezier(
    point: typing.Iterable[float | int],
    control_points: typing.Iterable[typing.Iterable[float | int]],
    round_to: float | int | None = 1e-6,
) -> bool:
    """Checks if a given point lies on the bezier curves with the given control points.

    This is done by solving the bezier polynomial with the point as the constant term; if
    any real roots exist, the point lies on the bezier curve.

    Parameters
    -----
    point
        The Cartesian Coordinates of the point to check.
    control_points
        The Cartesian Coordinates of the ordered control
        points of the bezier curve on which the point may
        or may not lie.
    round_to
        A float whose number of decimal places all values
        such as coordinates of points will be rounded.

    Returns
    -----
    bool
        Whether the point lies on the curve.
"""

    roots = proportions_along_bezier_curve_for_point(point, control_points, round_to)
    return len(roots) > 0

```

frame.py

```
"""Special rectangles."""

from __future__ import annotations

__all__ = [
    "ScreenRectangle",
    "FullScreenRectangle",
]
]

from manim.mobject.geometry.polygram import Rectangle
from .. import config

class ScreenRectangle(Rectangle):
    def __init__(self, aspect_ratio=16.0 / 9.0, height=4, **kwargs):
        super().__init__(width=aspect_ratio * height, height=height, **kwargs)

    @property
    def aspect_ratio(self):
        """The aspect ratio.

        When set, the width is stretched to accommodate
        the new aspect ratio.
        """

        return self.width / self.height

    @aspect_ratio.setter
    def aspect_ratio(self, value):
        self.stretch_to_fit_width(value * self.height)

class FullScreenRectangle(ScreenRectangle):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.height = config["frame_height"]
```

caching.py

```
from __future__ import annotations
from typing import Callable
from .. import config, logger
from ..utils.hashing import get_hash_from_play_call

def handle_caching_play(func: Callable[..., None]):
    """Decorator that returns a wrapped version of func that will compute
    the hash of the play invocation.

    The returned function will act according to the computed hash: either skip
    the animation because it's already cached, or let the invoked function
    play normally.

    Parameters
    -----
    func
        The play like function that has to be written to the video file stream.
        Take the same parameters as `scene.play`.
    """
    # NOTE : This is only kept for OpenGL renderer.
    # The play logic of the cairo renderer as been refactored and does not need this function anymore.
    # When OpenGL renderer will have a proper testing system,
    # the play logic of the latter has to be refactored in the same way the cairo renderer has been, and thus this
    # method has to be deleted.

    def wrapper(self, scene, *args, **kwargs):
        self.skip_animations = self._original_skipping_status
        self.update_skipping_status()
        animations = scene.compile_animations(*args, **kwargs)
        scene.add_mobjects_from_animations(animations)
        if self.skip_animations:
            logger.debug(f"Skipping animation {self.num_plays}")
            func(self, scene, *args, **kwargs)
            # If the animation is skipped, we mark its hash as None.
            # When sceneFileWriter will start combining partial movie files, it won't take into account None hashes.
            self.animations_hashes.append(None)
            self.file_writer.add_partial_movie_file(None)
            return
        if not config["disable_caching"]:
            mobjects_on_scene = scene.mobjects
            hash_play = get_hash_from_play_call(
                self,
                self.camera,
                animations,
                mobjects_on_scene,
            )
            if self.file_writer.is_already_cached(hash_play):
                logger.info(
                    f"Animation {self.num_plays} : Using cached data (hash : %(hash_play)s)",
                    {"hash_play": hash_play},
                )
                self.skip_animations = True
            else:
                hash_play = f"uncached_{self.num_plays:05}"
                self.animations_hashes.append(hash_play)
                self.file_writer.add_partial_movie_file(hash_play)
                logger.debug(
                    "List of the first few animation hashes of the scene: %(h)s",
                    {"h": str(self.animations_hashes[:5])},
                )
        func(self, scene, *args, **kwargs)

    return wrapper
```

constants.py

```
"""
Constant definitions.
"""

from __future__ import annotations

from enum import Enum

import numpy as np
from cloup import Context
from PIL.Image import Resampling

__all__ = [
    "SCENE_NOT_FOUND_MESSAGE",
    "CHOOSE_NUMBER_MESSAGE",
    "INVALID_NUMBER_MESSAGE",
    "NO_SCENE_MESSAGE",
    "NORMAL",
    "ITALIC",
    "OBLIQUE",
    "BOLD",
    "THIN",
    "ULTRALIGHT",
    "LIGHT",
    "SEMLIGHT",
    "BOOK",
    "MEDIUM",
    "SEMIBOLD",
    "ULTRABOLD",
    "HEAVY",
    "ULTRAHEAVY",
    "RESAMPLING_ALGORITHMS",
    "ORIGIN",
    "UP",
    "DOWN",
    "RIGHT",
    "LEFT",
    "IN",
    "OUT",
    "X_AXIS",
    "Y_AXIS",
    "Z_AXIS",
    "UL",
    "UR",
    "DL",
    "DR",
    "START_X",
    "START_Y",
    "DEFAULT_DOT_RADIUS",
    "DEFAULT_SMALL_DOT_RADIUS",
    "DEFAULT_DASH_LENGTH",
    "DEFAULT_ARROW_TIP_LENGTH",
    "SMALL_BUFF",
    "MED_SMALL_BUFF",
    "MED_LARGE_BUFF",
    "LARGE_BUFF",
    "DEFAULT_MOBJECT_TO_EDGE_BUFFER",
    "DEFAULT_MOBJECT_TO_MOBJECT_BUFFER",
    "DEFAULT_POINTWISE_FUNCTION_RUN_TIME",
    "DEFAULT_WAIT_TIME",
    "DEFAULT_POINT_DENSITY_2D",
    "DEFAULT_POINT_DENSITY_1D",
    "DEFAULT_STROKE_WIDTH",
    "DEFAULT_FONT_SIZE",
    "PI",
    "TAU",
    "DEGREES",
    "QUALITIES",
    "DEFAULT_QUALITY",
    "EPILOG",
    "CONTEXT_SETTINGS",
    "SHIFT_VALUE",
    "CTRL_VALUE",
    "RendererType",
    "LineJointType",
]
# Messages

SCENE_NOT_FOUND_MESSAGE: str = """
    {} is not in the script
"""

```

```

CHOOSE_NUMBER_MESSAGE: str = """
Choose number corresponding to desired scene/arguments.
(Use comma separated list for multiple entries)
Choice(s) : """

INVALID_NUMBER_MESSAGE: str = "Invalid scene numbers have been specified. Aborting."
NO_SCENE_MESSAGE: str = """
    There are no scenes inside that module
"""

# Pango stuff
NORMAL: str = "NORMAL"
ITALIC: str = "ITALIC"
OBLIQUE: str = "OBLIQUE"
BOLD: str = "BOLD"
# Only for Pango from below
THIN: str = "THIN"
ULTRALIGHT: str = "ULTRALIGHT"
LIGHT: str = "LIGHT"
SEMILIGHT: str = "SEMILIGHT"
BOOK: str = "BOOK"
MEDIUM: str = "MEDIUM"
SEMIBOLD: str = "SEMIBOLD"
ULTRABOLD: str = "ULTRABOLD"
HEAVY: str = "HEAVY"
ULTRAHEAVY: str = "ULTRAHEAVY"

RESAMPLING_ALGORITHMS = {
    "nearest": Resampling.NEAREST,
    "none": Resampling.NEAREST,
    "lanczos": Resampling.LANZOS,
    "antialias": Resampling.LANZOS,
    "bilinear": Resampling.BILINEAR,
    "linear": Resampling.BILINEAR,
    "bicubic": Resampling.BICUBIC,
    "cubic": Resampling.BICUBIC,
    "box": Resampling.BOX,
    "hamming": Resampling.HAMMING,
}

# Geometry: directions
ORIGIN: np.ndarray = np.array((0.0, 0.0, 0.0))
"""The center of the coordinate system."""

UP: np.ndarray = np.array((0.0, 1.0, 0.0))
"""One unit step in the positive Y direction."""

DOWN: np.ndarray = np.array((0.0, -1.0, 0.0))
"""One unit step in the negative Y direction."""

RIGHT: np.ndarray = np.array((1.0, 0.0, 0.0))
"""One unit step in the positive X direction."""

LEFT: np.ndarray = np.array((-1.0, 0.0, 0.0))
"""One unit step in the negative X direction."""

IN: np.ndarray = np.array((0.0, 0.0, -1.0))
"""One unit step in the negative Z direction."""

OUT: np.ndarray = np.array((0.0, 0.0, 1.0))
"""One unit step in the positive Z direction."""

# Geometry: axes
X_AXIS: np.ndarray = np.array((1.0, 0.0, 0.0))
Y_AXIS: np.ndarray = np.array((0.0, 1.0, 0.0))
Z_AXIS: np.ndarray = np.array((0.0, 0.0, 1.0))

# Geometry: useful abbreviations for diagonals
UL: np.ndarray = UP + LEFT
"""One step up plus one step left."""

UR: np.ndarray = UP + RIGHT
"""One step up plus one step right."""

DL: np.ndarray = DOWN + LEFT
"""One step down plus one step left."""

DR: np.ndarray = DOWN + RIGHT
"""One step down plus one step right."""

# Geometry
START_X: int = 30
START_Y: int = 20
DEFAULT_DOT_RADIUS: float = 0.08
DEFAULT_SMALL_DOT_RADIUS: float = 0.04
DEFAULT_DASH_LENGTH: float = 0.05
DEFAULT_ARROW_TIP_LENGTH: float = 0.35

```

```

# Default buffers (padding)
SMALL_BUFF: float = 0.1
MED_SMALL_BUFF: float = 0.25
MED_LARGE_BUFF: float = 0.5
LARGE_BUFF: float = 1
DEFAULT_MOBJECT_TO_EDGE_BUFFER: float = MED_LARGE_BUFF
DEFAULT_MOBJECT_TO_MOBJECT_BUFFER: float = MED_SMALL_BUFF

# Times in seconds
DEFAULT_POINTWISE_FUNCTION_RUN_TIME: float = 3.0
DEFAULT_WAIT_TIME: float = 1.0

# Misc
DEFAULT_POINT_DENSITY_2D: int = 25
DEFAULT_POINT_DENSITY_1D: int = 10
DEFAULT_STROKE_WIDTH: int = 4
DEFAULT_FONT_SIZE: float = 48

# Mathematical constants
PI: float = np.pi
"""The ratio of the circumference of a circle to its diameter."""

TAU: float = 2 * PI
"""The ratio of the circumference of a circle to its radius."""

DEGREES: float = TAU / 360
"""The exchange rate between radians and degrees."""

# Video qualities
QUALITIES: dict[str, dict[str, str | int | None]] = {
    "fourk_quality": {
        "flag": "k",
        "pixel_height": 2160,
        "pixel_width": 3840,
        "frame_rate": 60,
    },
    "production_quality": {
        "flag": "p",
        "pixel_height": 1440,
        "pixel_width": 2560,
        "frame_rate": 60,
    },
    "high_quality": {
        "flag": "h",
        "pixel_height": 1080,
        "pixel_width": 1920,
        "frame_rate": 60,
    },
    "medium_quality": {
        "flag": "m",
        "pixel_height": 720,
        "pixel_width": 1280,
        "frame_rate": 30,
    },
    "low_quality": {
        "flag": "l",
        "pixel_height": 480,
        "pixel_width": 854,
        "frame_rate": 15,
    },
    "example_quality": {
        "flag": None,
        "pixel_height": 480,
        "pixel_width": 854,
        "frame_rate": 30,
    },
},
}

DEFAULT_QUALITY: str = "high_quality"

EPILOG = "Made with <3 by Manim Community developers."
SHIFT_VALUE = 65505
CTRL_VALUE = 65507

CONTEXT_SETTINGS = Context.settings(
    align_option_groups=True,
    align_sections=True,
    show_constraints=True,
)

class RendererType(Enum):
    """An enumeration of all renderer types that can be assigned to
    the ``config.renderer`` attribute.

    Manim's configuration allows assigning string values to the renderer
    setting, the values are then replaced by the corresponding enum object.

```

In other words, you can run::

```
config.renderer = "opengl"
```

and checking the renderer afterwards reveals that the attribute has assumed the value::

```
<RendererType.OPENGL: 'opengl'>
"""
```

```
CAIRO = "cairo" #: A renderer based on the cairo backend.
OPENGL = "opengl" #: An OpenGL-based renderer.
```

```
class LineJointType(Enum):
    """Collection of available line joint types.
```

See the example below for a visual illustration of the different joint types.

Examples

```
.. manim:: LineJointVariants
:save_last_frame:

class LineJointVariants(Scene):
    def construct(self):
        mob = VMobject(stroke_width=20, color=GREEN).set_points_as_corners([
            np.array([-2, 0, 0]),
            np.array([0, 0, 0]),
            np.array([-2, 1, 0]),
        ])
        lines = VGroup(*[mob.copy() for _ in range(len(LineJointType))])
        for line, joint_type in zip(lines, LineJointType):
            line.joint_type = joint_type

        lines.arrange(RIGHT, buff=1)
        self.add(lines)
        for line in lines:
            label = Text(line.joint_type.name).next_to(line, DOWN)
            self.add(label)
"""

```

```
AUTO = 0
ROUND = 1
BEVEL = 2
MITER = 3
```

import_plugins.py

```
from __future__ import annotations
import types
import pkg_resources
from .. import config, logger
__all__ = []

plugins_requested: list = config["plugins"]
if "" in plugins_requested:
    plugins_requested.remove("")
for plugin in pkg_resources.iter_entry_points("manim.plugins"):
    if plugin.name not in plugins_requested:
        continue
    loaded_plugin = plugin.load()
    if isinstance(loaded_plugin, types.ModuleType):
        # it is a module so it can't be called
        # see if __all__ is defined
        # if it is defined use that to load all the modules necessary
        # essentially this would be similar to `from plugin import *`'
        # if not just import the module with the plugin name
        if hasattr(loaded_plugin, "__all__"):
            for thing in loaded_plugin.__all__: # type: ignore
                exec(f"{thing}={loaded_plugin.{thing}}")
            __all__.append(thing)
        else:
            exec(f"{plugin.name}={loaded_plugin}")
            __all__.append(plugin.name)
    elif isinstance(loaded_plugin, types.FunctionType):
        # call the function first
        # it will return a list of modules to add globally
        # finally add it
        lists = loaded_plugin()
        for lst in lists:
            exec(f"{lst.__name__}=lst")
            __all__.append(lst.__name__)
    plugins_requested.remove(plugin.name)

if plugins_requested != []:
    logger.warning("Missing Plugins: %s", plugins_requested)
```

tex.py

```
"""Utilities for processing LaTeX templates."""

from __future__ import annotations

__all__ = [
    "TexTemplate",
    "TexTemplateFromFile",
]

import copy
import os
import re
from pathlib import Path

class TexTemplate:
    """TeX templates are used for creating Tex() and MathTex() objects.

    Parameters
    -----
    tex_compiler : str
        The TeX compiler to be used, e.g. ``'latex'``, ``'pdflatex'`` or ``'lualatex'``
    output_format : str
        The output format resulting from compilation, e.g. ``'.dvi'`` or ``'.pdf'``
    documentclass : str
        The command defining the documentclass, e.g. ``'\documentclass[preview]{standalone}'``
    preamble : str
        The document's preamble, i.e. the part between ``'\documentclass'`` and ``'\begin{document}'``
    placeholder_text : str
        Text in the document that will be replaced by the expression to be rendered
    post_doc_commands : str
        Text (definitions, commands) to be inserted at right after ``'\begin{document}'``, e.g. ``'\boldsymbol{}```

    Attributes
    -----
    tex_compiler : :class:`str`
        The TeX compiler to be used, e.g. ``'latex'``, ``'pdflatex'`` or ``'lualatex'``
    output_format : :class:`str`
        The output format resulting from compilation, e.g. ``'.dvi'`` or ``'.pdf'``
    documentclass : :class:`str`
        The command defining the documentclass, e.g. ``'\documentclass[preview]{standalone}'``
    preamble : :class:`str`
        The document's preamble, i.e. the part between ``'\documentclass'`` and ``'\begin{document}'``
    placeholder_text : :class:`str`
        Text in the document that will be replaced by the expression to be rendered
    post_doc_commands : :class:`str`
        Text (definitions, commands) to be inserted at right after ``'\begin{document}'``, e.g. ``'\boldsymbol{}```

    default_documentclass = r"\documentclass[preview]{standalone}"
    default_preamble = r"""
\usepackage[english]{babel}
\usepackage{amsmath}
\usepackage{amsymb}
"""

    default_placeholder_text = "YourTextHere"
    default_tex_compiler = "latex"
    default_output_format = ".dvi"
    default_post_doc_commands = ""

    def __init__(self,
                 tex_compiler: str | None = None,
                 output_format: str | None = None,
                 documentclass: str | None = None,
                 preamble: str | None = None,
                 placeholder_text: str | None = None,
                 post_doc_commands: str | None = None,
                 **kwargs,
                 ):
        self.tex_compiler = (
            tex_compiler
            if tex_compiler is not None
            else TexTemplate.default_tex_compiler
        )
        self.output_format = (
            output_format
            if output_format is not None
            else TexTemplate.default_output_format
        )
        self.documentclass = (
            documentclass
            if documentclass is not None
            else TexTemplate.default_documentclass
        )
        self.preamble = (
            preamble
            if preamble is not None
            else TexTemplate.default_preamble
        )
        self.placeholder_text = (
            placeholder_text
            if placeholder_text is not None
            else TexTemplate.default_placeholder_text
        )
        self.post_doc_commands = (
            post_doc_commands
            if post_doc_commands is not None
            else TexTemplate.default_post_doc_commands
        )
        self._rebuild()

    def __eq__(self, other: TexTemplate) -> bool:
        return (
            self.body == other.body
            and self.tex_compiler == other.tex_compiler
            and self.output_format == other.output_format
            and self.post_doc_commands == other.post_doc_commands
        )

    def _rebuild(self):
        """Rebuilds the entire TeX template text from ``'\documentclass'`` to ``'\end{document}'`` according to all settings and choices."""
        self.body = (
            self.documentclass
            + "\n"
            + self.preamble
            + "\n"
            + r"\begin{document}"
            + "\n"
            + self.post_doc_commands
            + "\n"
            + self.placeholder_text
            + "\n"
            + r"\end{document}"
            + "\n"
        )

    def add_to_preamble(self, txt: str, prepend: bool = False):
        """Adds stuff to the TeX template's preamble (e.g. definitions, packages). Text can be inserted at the beginning or at the end of the preamble.

        Parameters
        -----
        txt : str
            String containing the text to be added, e.g. ``'\usepackage{hyperref}'``
```

```

prepend
    Whether the text should be added at the beginning of the preamble, i.e. right after ``\documentclass``. Default is to add it at the end of the preamble, i.e. right before
"""
if prepend:
    self.preamble = txt + "\n" + self.preamble
else:
    self.preamble += "\n" + txt
self._rebuild()

def add_to_document(self, txt: str):
    """Adds txt to the TeX template just after \begin{document}, e.g. ``\boldmath``

Parameters
-----
txt
    String containing the text to be added.
"""
self.post_doc_commands += "\n" + txt + "\n"
self._rebuild()

def get_texcode_for_expression(self, expression: str):
    """Inserts expression verbatim into TeX template.

Parameters
-----
expression
    The string containing the expression to be typeset, e.g. ``$\sqrt{2}$``

Returns
-----
:class:`str`
    LaTeX code based on current template, containing the given ``expression`` and ready for typesetting
"""
return self.body.replace(self.placeholder_text, expression)

def _texcode_for_environment(self, environment: str):
    """Processes the tex_environment string to return the correct ``\begin{environment}[extra]{extra}`` and ``\end{environment}`` strings

Parameters
-----
environment
    The tex_environment as a string. Acceptable formats include:
    ``{align*}``, ``align````, ``{tabular}[t]{cccl}``, ``{tabular}{cccl}``, ``\begin{tabular}[t]{cccl}``.

Returns
-----
Tuple[:class:`str`, :class:`str`]
    A pair of strings representing the opening and closing of the tex environment, e.g.
    ``\begin{tabular}{cccl}`` and ``\end{tabular}```
"""

# If the environment starts with \begin{, remove it
if environment[0:6] == r"\begin":
    environment = environment[6:]

# If environment begins with { strip it
if environment[0] == r"{":
    environment = environment[1:]

# The \begin command takes everything and closes with a brace
begin = r"\begin{" + environment
if (
    begin[-1] != r"}" and begin[-1] != r"]"
): # If it doesn't end on } or ], assume missing
    begin += r"}"

# While the \end command terminates at the first closing brace
split_at_brace = re.split(r"}", environment, 1)
end = r"\end{" + split_at_brace[0] + r"}"

return begin, end

def get_texcode_for_expression_in_env(self, expression: str, environment: str):
    """Inserts expression into TeX template wrapped in \begin{environment} and \end{environment}

Parameters
-----
expression
    The string containing the expression to be typeset, e.g. ``$\sqrt{2}$``
environment
    The string containing the environment in which the expression should be typeset, e.g. ``align``

Returns
-----
:class:`str`
    LaTeX code based on template, containing the given expression inside its environment, ready for typesetting
"""
begin, end = self._texcode_for_environment(environment)
return self.body.replace(self.placeholder_text, f"({begin}\n{expression}\n{end})")

def copy(self) -> TexTemplate:
    return copy.deepcopy(self)

class TexTemplateFromFile(TexTemplate):
    """A TexTemplate object created from a template file (default: tex_template.tex)

Parameters
-----
tex_filename
    Path to a valid TeX template file
kwargs
    Arguments for :class:`~.TexTemplate`.

Attributes
-----
template_file : :class:`str`
    Path to a valid TeX template file
body : :class:`str`
    Content of the TeX template file
tex_compiler : :class:`str`
    The TeX compiler to be used, e.g. ``latex```, ``pdflatex`` or ``lualatex```
output_format : :class:`str`
    The output format resulting from compilation, e.g. ``.dvi`` or ``.pdf``
"""

def __init__(self, *, tex_filename: str | os.PathLike = "tex_template.tex", **kwargs):
    self.template_file = Path(tex_filename)
    super().__init__(**kwargs)

def _rebuild(self):
    self.body = self.template_file.read_text()

def file_notMutable(self):
    raise Exception("Cannot modify TexTemplate when using a template file.")

def add_to_preamble(self, txt, prepend=False):
    self.file_notMutable()

def add_to_document(self, txt):
    self.file_notMutable()

```

commands.py_1

```
"""Manim's plugin subcommand.

Manim's plugin subcommand is accessed in the command-line interface via ``manim
plugin``. Here you can specify options, subcommands, and subgroups for the plugin
group.

"""

from __future__ import annotations

import cloup

from ...constants import CONTEXT_SETTINGS, EPILOG
from ...plugins.plugins_flags import list_plugins

@cloup.command(
    context_settings=CONTEXT_SETTINGS,
    no_args_is_help=True,
    epilog=EPILOG,
    help="Manages Manim plugins.",
)
@cloup.option(
    "-l",
    "--list",
    "list_available",
    is_flag=True,
    help="List available plugins.",
)
def plugins(list_available):
    if list_available:
        list_plugins()
```

ease_of_access_options.py

```
from __future__ import annotations

import click
from cloup import option, option_group

ease_of_access_options = option_group(
    "Ease of access options",
    option(
        "--progress_bar",
        default=None,
        show_default=False,
        type=click.Choice(
            ["display", "leave", "none"],
            case_sensitive=False,
        ),
        help="Display progress bars and/or keep them displayed.",
    ),
    option(
        "-p",
        "--preview",
        is_flag=True,
        help="Preview the Scene's animation. OpenGL does a live preview in a "
        "popup window. Cairo opens the rendered video file in the system "
        "default media player.",
        default=None,
    ),
    option(
        "-f",
        "--show_in_file_browser",
        is_flag=True,
        help="Show the output file in the file browser.",
        default=None,
    ),
    option(
        "--jupyter",
        is_flag=True,
        help="Using jupyter notebook magic.",
        default=None,
    ),
)
```

opengl_surface.py

```
from __future__ import annotations

from pathlib import Path
from typing import Iterable

import moderngl
import numpy as np
from PIL import Image

from manim.constants import *
from manim.mobject.opengl.opengl_mobject import OpenGLMobject
from manim.utils.bezier import integer_interpolate, interpolate
from manim.utils.color import *
from manim.utils.config_ops import _Data, _Uniforms
from manim.utils.deprecation import deprecated
from manim.utils.images import change_to_rgba_array, get_full_raster_image_path
from manim.utils.iterables import listify
from manim.utils.space_ops import normalize_along_axis

class OpenGLSurface(OpenGLMobject):
    r"""Creates a Surface.

    Parameters
    -----
    uv_func
        The function that defines the surface.
    u_range
        The range of the ``u`` variable: ````(u_min, u_max)````.
    v_range
        The range of the ``v`` variable: ````(v_min, v_max)````.
    resolution
        The number of samples taken of the surface.
    axes
        Axes on which the surface is to be drawn. Optional
        parameter used when coloring a surface by z-value.
    color
        Color of the surface. Defaults to grey.
    colorscale
        Colors of the surface. Optional parameter used when
        coloring a surface by values. Passing a list of
        colors and an axes will color the surface by z-value.
        Passing a list of tuples in the form ````(color, pivot)`````
        allows user-defined pivots where the color transitions.
    colorscale_axis
        Defines the axis on which the colorscale is applied
        (0 = x, 1 = y, 2 = z), default is z-axis (2).
    opacity
        Opacity of the surface from 0 being fully transparent
        to 1 being fully opaque. Defaults to 1.
    """

    shader_dtype = [
        ("point", np.float32, (3,)),
        ("du_point", np.float32, (3,)),
        ("dv_point", np.float32, (3,)),
        ("color", np.float32, (4,)),
    ]
    shader_folder = "surface"

    def __init__(
        self,
        uv_func=None,
        u_range=None,
        v_range=None,
        # Resolution counts number of points sampled, which for
        # each coordinate is one more than the the number of
        # rows/columns of approximating squares
        resolution=None,
        axes=None,
        color=GREY,
        colorscale=None,
        colorscale_axis=2,
        opacity=1.0,
        gloss=0.3,
        shadow=0.4,
        preferred_creation_axis=1,
        # For du and dv steps. Much smaller and numerical error
        # can crop up in the shaders.
        epsilon=1e-5,
        render_primitive=moderngl.TRIANGLES,
```

```

depth_test=True,
shader_folder=None,
**kwargs,
):
    self.passed_uv_func = uv_func
    self.u_range = u_range if u_range is not None else (0, 1)
    self.v_range = v_range if v_range is not None else (0, 1)
    # Resolution counts number of points sampled, which for
    # each coordinate is one more than the the number of
    # rows/columns of approximating squares
    self.resolution = resolution if resolution is not None else (101, 101)
    self.axes = axes
    self.colors = colors
    self.colors_axis = colors_axis
    self.preferred_creation_axis = preferred_creation_axis
    # For du and dv steps. Much smaller and numerical error
    # can crop up in the shaders.
    self.epsilon = epsilon

    self.triangle_indices = None
    super().__init__(
        color=color,
        opacity=opacity,
        gloss=gloss,
        shadow=shadow,
        shader_folder=shader_folder if shader_folder is not None else "surface",
        render_primitive=render_primitive,
        depth_test=depth_test,
        **kwargs,
    )
    self.compute_triangle_indices()

def uv_func(self, u, v):
    # To be implemented in subclasses
    if self.passed_uv_func:
        return self.passed_uv_func(u, v)
    return (u, v, 0.0)

def init_points(self):
    dim = self.dim
    nu, nv = self.resolution
    u_range = np.linspace(*self.u_range, nu)
    v_range = np.linspace(*self.v_range, nv)

    # Get three lists:
    # - Points generated by pure uv values
    # - Those generated by values nudged by du
    # - Those generated by values nudged by dv
    point_lists = []
    for du, dv in [(0, 0), (self.epsilon, 0), (0, self.epsilon)]:
        uv_grid = np.array([[u + du, v + dv] for v in v_range for u in u_range])
        point_grid = np.apply_along_axis(lambda p: self.uv_func(*p), 2, uv_grid)
        point_lists.append(point_grid.reshape((nu * nv, dim)))
    # Rather than tracking normal vectors, the points list will hold on to the
    # infinitesimal nudged values alongside the original values. This way, one
    # can perform all the manipulations they'd like to the surface, and normals
    # are still easily recoverable.
    self.set_points(np.vstack(point_lists))

def compute_triangle_indices(self):
    # TODO, if there is an event which changes
    # the resolution of the surface, make sure
    # this is called.
    nu, nv = self.resolution
    if nu == 0 or nv == 0:
        self.triangle_indices = np.zeros(0, dtype=int)
        return
    index_grid = np.arange(nu * nv).reshape((nu, nv))
    indices = np.zeros(6 * (nu - 1) * (nv - 1), dtype=int)
    indices[0::6] = index_grid[:-1, :-1].flatten() # Top left
    indices[1::6] = index_grid[+1:, :-1].flatten() # Bottom left
    indices[2::6] = index_grid[:-1, +1:].flatten() # Top right
    indices[3::6] = index_grid[+1:, +1:].flatten() # Top right
    indices[4::6] = index_grid[+1:, :-1].flatten() # Bottom left
    indices[5::6] = index_grid[+1:, +1:].flatten() # Bottom right
    self.triangle_indices = indices

def get_triangle_indices(self):
    return self.triangle_indices

def get_surface_points_and_nudged_points(self):
    points = self.points
    k = len(points) // 3
    return points[:k], points[k : 2 * k], points[2 * k :]

def get_unit_normals(self):
    s_points, du_points, dv_points = self.get_surface_points_and_nudged_points()

```

```

normals = np.cross(
    (du_points - s_points) / self.epsilon,
    (dv_points - s_points) / self.epsilon,
)
return normalize_along_axis(normals, 1)

def pointwise_become_partial(self, smobject, a, b, axis=None):
    assert isinstance(smobject, OpenGLSurface)
    if axis is None:
        axis = self.preferred_creation_axis
    if a <= 0 and b >= 1:
        self.match_points(smobject)
        return self

    nu, nv = smobject.resolution
    self.set_points(
        np.vstack(
            [
                self.get_partial_points_array(
                    arr.copy(),
                    a,
                    b,
                    (nu, nv, 3),
                    axis=axis,
                )
                for arr in smobject.get_surface_points_and_nudged_points()
            ],
        ),
    )
    return self

def get_partial_points_array(self, points, a, b, resolution, axis):
    if len(points) == 0:
        return points
    nu, nv = resolution[:2]
    points = points.reshape(resolution)
    max_index = resolution[axis] - 1
    lower_index, lower_residue = integer_interpolate(0, max_index, a)
    upper_index, upper_residue = integer_interpolate(0, max_index, b)
    if axis == 0:
        points[:lower_index] = interpolate(
            points[lower_index],
            points[lower_index + 1],
            lower_residue,
        )
        points[upper_index + 1 :] = interpolate(
            points[upper_index],
            points[upper_index + 1],
            upper_residue,
        )
    else:
        shape = (nu, 1, resolution[2])
        points[:, :lower_index] = interpolate(
            points[:, lower_index],
            points[:, lower_index + 1],
            lower_residue,
        ).reshape(shape)
        points[:, upper_index + 1 :] = interpolate(
            points[:, upper_index],
            points[:, upper_index + 1],
            upper_residue,
        ).reshape(shape)
    return points.reshape((nu * nv, *resolution[2:]))

def sort_faces_back_to_front(self, vect=OUT):
    tri_is = self.triangle_indices
    indices = list(range(len(tri_is) // 3))
    points = self.points

    def index_dot(index):
        return np.dot(points[tri_is[3 * index]], vect)

    indices.sort(key=index_dot)
    for k in range(3):
        tri_is[k::3] = tri_is[k::3][indices]
    return self

# For shaders
def get_shader_data(self):
    """Called by parent Mobject to calculate and return
    the shader data.

    Returns
    ------
    shader_dtype
        An array containing the shader data (vertices and
        color of each vertex)
    """

```

```

"""
s_points, du_points, dv_points = self.get_surface_points_and_nudged_points()
shader_data = np.zeros(len(s_points), dtype=self.shader_dtype)
if "points" not in self.locked_data_keys:
    shader_data["point"] = s_points
    shader_data["du_point"] = du_points
    shader_data["dv_point"] = dv_points
    if self.colorscales:
        if not hasattr(self, "color_by_val"):
            self.color_by_val = self._get_color_by_value(s_points)
        shader_data["color"] = self.color_by_val
    else:
        self.fill_in_shader_color_info(shader_data)
return shader_data

def fill_in_shader_color_info(self, shader_data):
    """Fills in the shader color data when the surface
    is all one color.

    Parameters
    -----
    shader_data
        The vertices of the surface.

    Returns
    -----
    shader_dtype
        An array containing the shader data (vertices and
        color of each vertex)
    """
    self.read_data_to_shader(shader_data, "color", "rgbas")
    return shader_data

def _get_color_by_value(self, s_points):
    """Matches each vertex to a color associated to it's z-value.

    Parameters
    -----
    s_points
        The vertices of the surface.

    Returns
    -----
    List
        A list of colors matching the vertex inputs.
    """
    if type(self.colorscales[0]) in (list, tuple):
        new_colors, pivots = [
            [i for i, j in self.colorscales],
            [j for i, j in self.colorscales],
        ]
    else:
        new_colors = self.colorscales

    pivot_min = self.axes.z_range[0]
    pivot_max = self.axes.z_range[1]
    pivot_frequency = (pivot_max - pivot_min) / (len(new_colors) - 1)
    pivots = np.arange(
        start=pivot_min,
        stop=pivot_max + pivot_frequency,
        step=pivot_frequency,
    )

    return_colors = []
    for point in s_points:
        axis_value = self.axes.point_to_coords(point)[self.colorscales_axis]
        if axis_value <= pivots[0]:
            return_colors.append(color_to_rgba(new_colors[0], self.opacity))
        elif axis_value >= pivots[-1]:
            return_colors.append(color_to_rgba(new_colors[-1], self.opacity))
        else:
            for i, pivot in enumerate(pivots):
                if pivot > axis_value:
                    color_index = (axis_value - pivots[i - 1]) / (
                        pivots[i] - pivots[i - 1]
                    )
                    color_index = max(min(color_index, 1), 0)
                    temp_color = interpolate_color(
                        new_colors[i - 1],
                        new_colors[i],
                        color_index,
                    )
                    break
            return_colors.append(color_to_rgba(temp_color, self.opacity))

    return return_colors

```

```

def get_shader_vert_indices(self):
    return self.get_triangle_indices()

class OpenGLSurfaceGroup(OpenGLSurface):
    def __init__(self, *parametric_surfaces, resolution=None, **kwargs):
        self.resolution = (0, 0) if resolution is None else resolution
        super().__init__(uv_func=None, **kwargs)
        self.add(*parametric_surfaces)

    def init_points(self):
        pass # Needed?

class OpenGLTexturedSurface(OpenGLSurface):
    shader_dtype = [
        ("point", np.float32, (3,)),
        ("du_point", np.float32, (3,)),
        ("dv_point", np.float32, (3,)),
        ("im_coords", np.float32, (2,)),
        ("opacity", np.float32, (1,)),
    ]
    shader_folder = "textured_surface"
    im_coords = _Data()
    opacity = _Data()
    num_textures = _Uniforms()

    def __init__(
        self,
        uv_surface: OpenGLSurface,
        image_file: str | Path,
        dark_image_file: str | Path = None,
        image_mode: str | Iterable[str] = "RGBA",
        shader_folder: str | Path = None,
        **kwargs,
    ):
        self.uniforms = {}

        if not isinstance(uv_surface, OpenGLSurface):
            raise Exception("uv_surface must be of type OpenGLSurface")
        if type(image_file) == np.ndarray:
            image_file = change_to_rgba_array(image_file)

        # Set texture information
        if isinstance(image_mode, (str, Path)):
            image_mode = [image_mode] * 2
        image_mode_light, image_mode_dark = image_mode
        texture_paths = {
            "LightTexture": self.get_image_from_file(
                image_file,
                image_mode_light,
            ),
            "DarkTexture": self.get_image_from_file(
                dark_image_file or image_file,
                image_mode_dark,
            ),
        }
        if dark_image_file:
            self.num_textures = 2

        self.uv_surface = uv_surface
        self.uv_func = uv_surface.uv_func
        self.u_range = uv_surface.u_range
        self.v_range = uv_surface.v_range
        self.resolution = uv_surface.resolution
        self.gloss = self.uv_surface.gloss
        super().__init__(texture_paths=texture_paths, **kwargs)

    def get_image_from_file(
        self,
        image_file: str | Path,
        image_mode: str,
    ):
        image_file = get_full_raster_image_path(image_file)
        return Image.open(image_file).convert(image_mode)

    def init_data(self):
        super().init_data()
        self.im_coords = np.zeros((0, 2))
        self.opacity = np.zeros((0, 1))

    def init_points(self):
        nu, nv = self.uv_surface.resolution
        self.set_points(self.uv_surface.points)
        self.im_coords = np.array([
            [u, v]

```

```

        for u in np.linspace(0, 1, nu)
        for v in np.linspace(1, 0, nv) # Reverse y-direction
    ],
)

def init_colors(self):
    self.opacity = np.array([self.uv_surface.rgbas[:, 3]])

def set_opacity(self, opacity, recurse=True):
    for mob in self.get_family(recurse):
        mob.opacity = np.array([[o] for o in listify(opacity)])
    return self

def pointwise_become_partial(self, tsmobject, a, b, axis=1):
    super().pointwise_become_partial(tsmobject, a, b, axis)
    im_coords = self.im_coords
    im_coords[:] = tsmobject.im_coords
    if a <= 0 and b >= 1:
        return self
    nu, nv = tsmobject.resolution
    im_coords[:] = self.get_partial_points_array(im_coords, a, b, (nu, nv, 2), axis)
    return self

def fill_in_shader_color_info(self, shader_data):
    self.read_data_to_shader(shader_data, "opacity", "opacity")
    self.read_data_to_shader(shader_data, "im_coords", "im_coords")
    return shader_data

```

number_line.py

```
"""Mobject representing a number line."""
from __future__ import annotations

from manim.mobject.opengl.opengl_vectorized_mobject import OpenGLVMobject
__all__ = ["NumberLine", "UnitInterval"]

from typing import TYPE_CHECKING, Callable, Iterable, Sequence

if TYPE_CHECKING:
    from manim.mobject.geometry.tips import ArrowTip

import numpy as np

from manim import config
from manim.constants import *
from manim.mobject.geometry.line import Line
from manim.mobject.graphing.scale import LinearBase, _ScaleBase
from manim.mobject.text.numbers import DecimalNumber
from manim.mobject.tex.tex_mobject import MathTex, Tex
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.bezier import interpolate
from manim.utils.config_ops import merge_dicts_recursively
from manim.utils.space_ops import normalize

class NumberLine(Line):
    """Creates a number line with tick marks.

    Parameters
    -----
    x_range
        The ``[x_min, x_max, x_step]`` values to create the line.
    length
        The length of the number line.
    unit_size
        The distance between each tick of the line. Overwritten by :attr:`length`, if specified.
    include_ticks
        Whether to include ticks on the number line.
    tick_size
        The length of each tick mark.
    numbers_with_elongated_ticks
        An iterable of specific values with elongated ticks.
    longer_tick_multiple
        Influences how many times larger elongated ticks are than regular ticks ( $2 = 2x$ ).
    rotation
        The angle (in radians) at which the line is rotated.
    stroke_width
        The thickness of the line.
    include_tip
        Whether to add a tip to the end of the line.
    tip_width
        The width of the tip.
    tip_height
        The height of the tip.
    tip_shape
        The mobject class used to construct the tip, or ``None`` (the default) for the default arrow tip. Passed classes have to inherit from :class:`.ArrowTip`.
    include_numbers
        Whether to add numbers to the tick marks. The number of decimal places is determined by the step size, this default can be overridden by ``decimal_number_config``.
    scaling
        The way the ``x_range`` is value is scaled, i.e. :class:`~.LogBase` for a logarithmic numberline. Defaults to :class:`~.LinearBase`.
    font_size
        The size of the label mobjects. Defaults to 36.
    label_direction
        The specific position to which label mobjects are added on the line.
    label_constructor
        Determines the mobject class that will be used to construct the labels of the number line.
    line_to_number_buff
        The distance between the line and the label mobject.
    decimal_number_config
        Arguments that can be passed to :class:`~.numbers.DecimalNumber` to influence number mobjects.
    numbers_to_exclude
        An explicit iterable of numbers to not be added to the number line.
    numbers_to_include
        An explicit iterable of numbers to add to the number line
    kwargs
        Additional arguments to be passed to :class:`~.Line`.

    .. note::
        Number ranges that include both negative and positive values will be generated from the 0 point, and may not include a tick at the min / max values as the tick locations are dependent on the step size.

    Examples
    -----
    .. manim:: NumberLineExample
        :save_last_frame:

        class NumberLineExample(Scene):
            def construct(self):
                10 = NumberLine(
                    x_range=[-10, 10, 2],
                    length=10,
                    color=BLUE,
```

```

        include_numbers=True,
        label_direction=UP,
    )

    11 = NumberLine(
        x_range=[-10, 10, 2],
        unit_size=0.5,
        numbers_with_elongated_ticks=[-2, 4],
        include_numbers=True,
        font_size=24,
    )
    num6 = 11.numbers[8]
    num6.set_color(RED)

    12 = NumberLine(
        x_range=[-2.5, 2.5 + 0.5, 0.5],
        length=12,
        decimal_number_config={"num_decimal_places": 2},
        include_numbers=True,
    )

    13 = NumberLine(
        x_range=[-5, 5 + 1, 1],
        length=6,
        include_tip=True,
        include_numbers=True,
        rotation=10 * DEGREES,
    )

    line_group = VGroup(10, 11, 12, 13).arrange(DOWN, buff=1)
    self.add(line_group)
"""

def __init__(
    self,
    x_range: Sequence[float] | None = None, # must be first
    length: float | None = None,
    unit_size: float = 1,
    # ticks
    include_ticks: bool = True,
    tick_size: float = 0.1,
    numbers_with_elongated_ticks: Iterable[float] | None = None,
    longer_tick_multiple: int = 2,
    exclude_origin_tick: bool = False,
    # visuals
    rotation: float = 0,
    stroke_width: float = 2.0,
    # tip
    include_tip: bool = False,
    tip_width: float = DEFAULT_ARROW_TIP_LENGTH,
    tip_height: float = DEFAULT_ARROW_TIP_LENGTH,
    tip_shape: type[ArrowTip] | None = None,
    # numbers/labels
    include_numbers: bool = False,
    font_size: float = 36,
    label_direction: Sequence[float] = DOWN,
    label_constructor: VMobject = MathTex,
    scaling: _ScaleBase = LinearBase(),
    line_to_number_buff: float = MED_SMALL_BUFF,
    decimal_number_config: dict | None = None,
    numbers_to_exclude: Iterable[float] | None = None,
    numbers_to_include: Iterable[float] | None = None,
    **kwargs,
):
    # avoid mutable arguments in defaults
    if numbers_to_exclude is None:
        numbers_to_exclude = []
    if numbers_with_elongated_ticks is None:
        numbers_with_elongated_ticks = []

    if x_range is None:
        x_range = [
            round(-config["frame_x_radius"]),
            round(config["frame_x_radius"]),
            1,
        ]
    elif len(x_range) == 2:
        # adds x_step if not specified. not sure how to feel about this. a user can't know default without peeking at source code
        x_range = [*x_range, 1]

    if decimal_number_config is None:
        decimal_number_config = {
            "num_decimal_places": self._decimal_places_from_step(x_range[2]),
        }

    # turn into a NumPy array to scale by just applying the function
    self.x_range = np.array(x_range, dtype=float)
    self.x_min, self.x_max, self.x_step = scaling.function(self.x_range)
    self.length = length
    self.unit_size = unit_size
    # ticks
    self.include_ticks = include_ticks
    self.tick_size = tick_size
    self.numbers_with_elongated_ticks = numbers_with_elongated_ticks
    self.longer_tick_multiple = longer_tick_multiple
    self.exclude_origin_tick = exclude_origin_tick
    # visuals
    self.rotation = rotation
    # tip
    self.include_tip = include_tip
    self.tip_width = tip_width
    self.tip_height = tip_height
    # numbers
    self.font_size = font_size
    self.include_numbers = include_numbers
    self.label_direction = label_direction
    self.label_constructor = label_constructor

```

```

self.line_to_number_buff = line_to_number_buff
self.decimal_number_config = decimal_number_config
self.numbers_to_exclude = numbers_to_exclude
self.numbers_to_include = numbers_to_include

self.scaling = scaling
super().__init__(
    self.x_range[0] * RIGHT,
    self.x_range[1] * RIGHT,
    stroke_width=stroke_width,
    **kwargs,
)
if self.length:
    self.set_length(self.length)
    self.unit_size = self.get_unit_size()
else:
    self.scale(self.unit_size)

self.center()

if self.include_tip:
    self.add_tip(
        tip_length=self.tip_height,
        tip_width=self.tip_width,
        tip_shape=tip_shape,
    )
    self.tip.set_stroke(self.stroke_color, self.stroke_width)

if self.include_ticks:
    self.add_ticks()

self.rotate(self.rotation)
if self.include_numbers or self.numbers_to_include is not None:
    if self.scaling.custom_labels:
        tick_range = self.get_tick_range()

        self.add_labels(
            dict(
                zip(
                    tick_range,
                    self.scaling.get_custom_labels(
                        tick_range,
                        unit_decimal_places=decimal_number_config[
                            "num_decimal_places"
                        ],
                    ),
                ),
            ),
        )
    else:
        self.add_numbers(
            x_values=self.numbers_to_include,
            excluding=self.numbers_to_exclude,
            font_size=self.font_size,
        )

def rotate_about_zero(self, angle: float, axis: Sequence[float] = OUT, **kwargs):
    return self.rotate_about_number(0, angle, axis, **kwargs)

def rotate_about_number(
    self, number: float, angle: float, axis: Sequence[float] = OUT, **kwargs
):
    return self.rotate(angle, axis, about_point=self.n2p(number), **kwargs)

def add_ticks(self):
    """Adds ticks to the number line. Ticks can be accessed after creation
    via ``self.ticks``."""
    ticks = VGroup()
    elongated_tick_size = self.tick_size * self.longer_tick_multiple
    for x in self.get_tick_range():
        size = self.tick_size
        if x in self.numbers_with_elongated_ticks:
            size = elongated_tick_size
        ticks.add(self.get_tick(x, size))
    self.add(ticks)
    self.ticks = ticks

def get_tick(self, x: float, size: float | None = None) -> Line:
    """Generates a tick and positions it along the number line.

    Parameters
    -----
    x
        The position of the tick.
    size
        The factor by which the tick is scaled.

    Returns
    -----
    :class:`~.Line`
        A positioned tick.
    """
    if size is None:
        size = self.tick_size
    result = Line(size * DOWN, size * UP)
    result.rotate(self.get_angle())
    result.move_to(self.number_to_point(x))
    result.match_style(self)
    return result

def get_tick_marks(self) -> VGroup:
    return self.ticks

def get_tick_range(self) -> np.ndarray:
    """Generates the range of values on which labels are plotted based on the

```

```

``x_range`` attribute of the number line.

Returns
-----
np.ndarray
    A numpy array of floats representing values along the number line.
"""

x_min, x_max, x_step = self.x_range
if not self.include_tip:
    x_max += 1e-6

# Handle cases where min and max are both positive or both negative
if x_min < x_max < 0 or x_max > x_min > 0:
    tick_range = np.arange(x_min, x_max, x_step)
else:
    start_point = 0
    if self.exclude_origin_tick:
        start_point += x_step

    x_min_segment = np.arange(start_point, np.abs(x_min) + 1e-6, x_step) * -1
    x_max_segment = np.arange(start_point, x_max, x_step)

    tick_range = np.unique(np.concatenate((x_min_segment, x_max_segment)))

return self.scaling.function(tick_range)

def number_to_point(self, number: float | np.ndarray) -> np.ndarray:
    """Accepts a value along the number line and returns a point with
    respect to the scene.

    Parameters
    -----
    number
        The value to be transformed into a coordinate. Or a list of values.

    Returns
    -----
    np.ndarray
        A point with respect to the scene's coordinate system. Or a list of points.

    Examples
    -----
    >>> from manim import NumberLine
    >>> number_line = NumberLine()
    >>> number_line.number_to_point(0)
    array([0., 0., 0.])
    >>> number_line.number_to_point(1)
    array([1., 0., 0.])
    >>> number_line.number_to_point([1,2,3])
    array([[1., 0., 0.],
           [2., 0., 0.],
           [3., 0., 0.]])
    """
    number = np.asarray(number)
    scalar = number.ndim == 0
    number = self.scaling.inverse_function(number)
    alphas = (number - self.x_range[0]) / (self.x_range[1] - self.x_range[0])
    alphas = float(alphas) if scalar else np.vstack(alphas)
    val = interpolate(self.get_start(), self.get_end(), alphas)
    return val

def point_to_number(self, point: Sequence[float]) -> float:
    """Accepts a point with respect to the scene and returns
    a float along the number line.

    Parameters
    -----
    point
        A sequence of values consisting of ``(x_coord, y_coord, z_coord)``.

    Returns
    -----
    float
        A float representing a value along the number line.

    Examples
    -----
    >>> from manim import NumberLine
    >>> number_line = NumberLine()
    >>> number_line.point_to_number((0,0,0))
    0.0
    >>> number_line.point_to_number((1,0,0))
    1.0
    >>> number_line.point_to_number(([0.5,0,0],[1,0,0],[1.5,0,0]))
    array([0.5, 1., 1.5])

    """
    point = np.asarray(point)
    start, end = self.get_start_and_end()
    unit_vect = normalize(end - start)
    proportion = np.dot(point - start, unit_vect) / np.dot(end - start, unit_vect)
    return interpolate(self.x_min, self.x_max, proportion)

def n2p(self, number: float | np.ndarray) -> np.ndarray:
    """Abbreviation for :meth:`~.NumberLine.number_to_point`."""
    return self.number_to_point(number)

def p2n(self, point: Sequence[float]) -> float:
    """Abbreviation for :meth:`~.NumberLine.point_to_number`."""
    return self.point_to_number(point)

def get_unit_size(self) -> float:
    return self.get_length() / (self.x_range[1] - self.x_range[0])

def get_unit_vector(self) -> np.ndarray:

```

```

    return super().get_unit_vector() * self.unit_size

def get_number_mobject(
    self,
    x: float,
    direction: Sequence[float] | None = None,
    buff: float | None = None,
    font_size: float | None = None,
    label_constructor: VMobject | None = None,
    **number_config,
) -> VMobject:
    """Generates a positioned :class:`~.DecimalNumber` mobject
    generated according to ``label_constructor``.

    Parameters
    -----
    x
        The x-value at which the mobject should be positioned.
    direction
        Determines the direction at which the label is positioned next to the line.
    buff
        The distance of the label from the line.
    font_size
        The font size of the label mobject.
    label_constructor
        The :class:`~.VMobject` class that will be used to construct the label.
        Defaults to the ``label_constructor`` attribute of the number line
        if not specified.

    Returns
    -----
    :class:`~.DecimalNumber`
        The positioned mobject.
    """
    number_config = merge_dicts_recursively(
        self.decimal_number_config,
        number_config,
    )
    if direction is None:
        direction = self.label_direction
    if buff is None:
        buff = self.line_to_number_buff
    if font_size is None:
        font_size = self.font_size
    if label_constructor is None:
        label_constructor = self.label_constructor

    num_mob = DecimalNumber(
        x, font_size=font_size, mob_class=label_constructor, **number_config
    )

    num_mob.next_to(self.number_to_point(x), direction=direction, buff=buff)
    if x < 0 and self.label_direction[0] == 0:
        # Align without the minus sign
        num_mob.shift(num_mob[0].get_width() * LEFT / 2)
    return num_mob

def get_number_mobjects(self, *numbers, **kwargs) -> VGroup:
    if len(numbers) == 0:
        numbers = self.default_numbers_to_display()
    return VGroup(*[self.get_number_mobject(number, **kwargs) for number in numbers])

def get_labels(self) -> VGroup:
    return self.get_number_mobjects()

def add_numbers(
    self,
    x_values: Iterable[float] | None = None,
    excluding: Iterable[float] | None = None,
    font_size: float | None = None,
    label_constructor: VMobject | None = None,
    **kwargs,
):
    """Adds :class:`~.DecimalNumber` mobjects representing their position
    at each tick of the number line. The numbers can be accessed after creation
    via ``self.numbers``.

    Parameters
    -----
    x_values
        An iterable of the values used to position and create the labels.
        Defaults to the output produced by :meth:`~.NumberLine.get_tick_range`.
    excluding
        A list of values to exclude from :attr:`x_values`.
    font_size
        The font size of the labels. Defaults to the ``font_size`` attribute
        of the number line.
    label_constructor
        The :class:`~.VMobject` class that will be used to construct the label.
        Defaults to the ``label_constructor`` attribute of the number line
        if not specified.
    """
    if x_values is None:
        x_values = self.get_tick_range()

    if excluding is None:
        excluding = self.numbers_to_exclude

    if font_size is None:
        font_size = self.font_size

    if label_constructor is None:
        label_constructor = self.label_constructor

    numbers = VGroup()
    for x in x_values:
        if x in excluding:

```

```

        continue
    numbers.add(
        self.get_number_mobject(
            x,
            font_size=font_size,
            label_constructor=label_constructor,
            **kwargs,
        )
    )
self.add(numbers)
self.numbers = numbers
return self

def add_labels(
    self,
    dict_values: dict[float, str | float | VMobject],
    direction: Sequence[float] = None,
    buff: float | None = None,
    font_size: float | None = None,
    label_constructor: VMobject | None = None,
):
    """Adds specifically positioned labels to the :class:`~.NumberLine` using a ``dict``.
    The labels can be accessed after creation via ``self.labels``.

    Parameters
    -----
    dict_values
        A dictionary consisting of the position along the number line and the mobject to be added:
        ``{1: Tex("Monday"), 3: Tex("Tuesday")}``. :attr:`label_constructor` will be used
        to construct the labels if the value is not a mobject (``str`` or ``float``).
    direction
        Determines the direction at which the label is positioned next to the line.
    buff
        The distance of the label from the line.
    font_size
        The font size of the mobject to be positioned.
    label_constructor
        The :class:`~.VMobject` class that will be used to construct the label.
        Defaults to the ``label_constructor`` attribute of the number line
        if not specified.

    Raises
    -----
    AttributeError
        If the label does not have a ``font_size`` attribute, an ``AttributeError`` is raised.
    """

    direction = self.label_direction if direction is None else direction
    buff = self.line_to_number_buff if buff is None else buff
    font_size = self.font_size if font_size is None else font_size
    if label_constructor is None:
        label_constructor = self.label_constructor

    labels = VGroup()
    for x, label in dict_values.items():
        # TODO: remove this check and ability to call
        # this method via CoordinateSystem.add_coordinates()
        # must be explicitly called
        if isinstance(label, str) and label_constructor is MathTex:
            label = Tex(label)
        else:
            label = self._create_label_tex(label, label_constructor)

        if hasattr(label, "font_size"):
            label.font_size = font_size
        else:
            raise AttributeError(f"(label) is not compatible with add_labels.")
        label.next_to(self.number_to_point(x), direction=direction, buff=buff)
        labels.add(label)

    self.labels = labels
    self.add(labels)
    return self

def _create_label_tex(
    self,
    label_tex: str | float | VMobject,
    label_constructor: Callable | None = None,
    **kwargs,
) -> VMobject:
    """Checks if the label is a :class:`~.VMobject`, otherwise, creates a
    label by passing ``label_tex`` to ``label_constructor``.

    Parameters
    -----
    label_tex
        The label for which a mobject should be created. If the label already
        is a mobject, no new mobject is created.
    label_constructor
        Optional. A class or function returning a mobject when
        passing ``label_tex`` as an argument. If ``None`` is passed
        (the default), the label constructor from the :attr:`.label_constructor`
        attribute is used.

    Returns
    -----
    :class:`~.VMobject`
        The label.

    """
    if label_constructor is None:
        label_constructor = self.label_constructor
    if isinstance(label_tex, (VMobject, OpenGLMobject)):
        return label_tex
    else:
        return label_constructor(label_tex, **kwargs)

@staticmethod

```

```
def _decimal_places_from_step(step) -> int:
    step = str(step)
    if "." not in step:
        return 0
    return len(step.split(".")[-1])

class UnitInterval(NumberLine):
    def __init__(self,
                 unit_size=10,
                 numbers_with_elongated_ticks=None,
                 decimal_number_config=None,
                 **kwargs,
                 ):
        numbers_with_elongated_ticks = (
            [0, 1]
            if numbers_with_elongated_ticks is None
            else numbers_with_elongated_ticks
        )

        decimal_number_config = (
            {
                "num_decimal_places": 1,
            }
            if decimal_number_config is None
            else decimal_number_config
        )

        super().__init__(
            x_range=(0, 1, 0.1),
            unit_size=unit_size,
            numbers_with_elongated_ticks=numbers_with_elongated_ticks,
            decimal_number_config=decimal_number_config,
            **kwargs,
        )
```

boolean_ops.py

```
"""Boolean operations for two-dimensional mobjects."""

from __future__ import annotations

import typing

import numpy as np
from pathops import Path as SkiaPath
from pathops import PathVerb, difference, intersection, union, xor

from manim import config
from manim.mobject.opengl_compatibility import ConvertToOpenGL
from manim.mobject.types.vectorized_mobject import VMobject

from ...constants import RendererType

__all__ = ["Union", "Intersection", "Difference", "Exclusion"]

class _BooleanOps(VMobject, metaclass=ConvertToOpenGL):
    """This class contains some helper functions which
    helps to convert to and from skia objects and manim
    objects (:class:`~.VMobject`).

    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def _convert_2d_to_3d_array(
        self,
        points: typing.Iterable,
        z_dim: float = 0.0,
    ) -> list[np.ndarray]:
        """Converts an iterable with coordinates in 2d to 3d by adding
        :attr:`z_dim` as the z coordinate.

        Parameters
        -----
        points:
            An iterable which has the coordinates.
        z_dim:
            The default value of z coordinate.

        Returns
        -----
        typing.List[np.ndarray]
            A list of array converted to 3d.

        Example
        -----
        >>> a = _BooleanOps()
        >>> p = [(1, 2), (3, 4)]
        >>> a._convert_2d_to_3d_array(p)
        [array([1., 2., 0.]), array([3., 4., 0.])]

        points = list(points)
        for i, point in enumerate(points):
            if len(point) == 2:
                points[i] = np.array(list(point) + [z_dim])
        return points

    def _convert_vmobject_to_skia_path(self, vmobject: VMobject) -> SkiaPath:
        """Converts a :class:`~.VMobject` to SkiaPath. This method only works for
        cairo renderer because it treats the points as Cubic beizer curves.

        Parameters
        -----
        vmobject:
            The :class:`~.VMobject` to convert from.

        Returns
        -----
        SkiaPath:
            The converted path.
        """

        path = SkiaPath()

        if not np.all(np.isfinite(vmobject.points)):
            points = np.zeros((1, 3)) # point invalid?
        else:
            points = vmobject.points
```

```

if len(points) == 0: # what? No points so return empty path
    return path

# In OpenGL it's quadratic bezier curves while on Cairo it's cubic...
if config.renderer == RendererType.OPENGL:
    subpaths = vmobject.get_subpaths_from_points(points)
    for subpath in subpaths:
        quads = vmobject.get_bezier_tuples_from_points(subpath)
        start = subpath[0]
        path.moveTo(*start[:2])
        for p0, p1, p2 in quads:
            path.quadTo(*p1[:2], *p2[:2])
        if vmobject.consider_points_equals(subpath[0], subpath[-1]):
            path.close()
elif config.renderer == RendererType.CAIRO:
    subpaths = vmobject.gen_subpaths_from_points_2d(points)
    for subpath in subpaths:
        quads = vmobject.gen_cubic_bezier_tuples_from_points(subpath)
        start = subpath[0]
        path.moveTo(*start[:2])
        for p0, p1, p2, p3 in quads:
            path.cubicTo(*p1[:2], *p2[:2], *p3[:2])

        if vmobject.consider_points_equals_2d(subpath[0], subpath[-1]):
            path.close()

return path

def _convert_skia_path_to_vmobject(self, path: SkiaPath) -> VMobject:
    """Converts SkiaPath back to VMobject.

    Parameters
    -----
    path:
        The SkiaPath to convert.

    Returns
    -----
    VMobject:
        The converted VMobject.
    """
    vmobject = self
    current_path_start = np.array([0, 0, 0])

    for path_verb, points in path:
        if path_verb == PathVerb.MOVE:
            parts = self._convert_2d_to_3d_array(points)
            for part in parts:
                current_path_start = part
                vmobject.start_new_path(part)
                # vmobject.move_to(*part)
        elif path_verb == PathVerb.CUBIC:
            n1, n2, n3 = self._convert_2d_to_3d_array(points)
            vmobject.add_cubic_bezier_curve_to(n1, n2, n3)
        elif path_verb == PathVerb.LINE:
            parts = self._convert_2d_to_3d_array(points)
            vmobject.add_line_to(parts[0])
        elif path_verb == PathVerb CLOSE:
            vmobject.add_line_to(current_path_start)
        elif path_verb == PathVerb.QUAD:
            n1, n2 = self._convert_2d_to_3d_array(points)
            vmobject.add_quadratic_bezier_curve_to(n1, n2)
        else:
            raise Exception("Unsupported: %s" % path_verb)
    return vmobject

class Union(_BooleanOps):
    """Union of two or more :class:`~.VMobject`'s. This returns the common region of
    the :class:`~VMobject`'s.

    Parameters
    -----
    vmobjects
        The :class:`~.VMobject`'s to find the union of.

    Raises
    -----
    ValueError
        If less than 2 :class:`~.VMobject`'s are passed.

    Example
    -----
    .. manim:: UnionExample
        :save_last_frame:

    class UnionExample(Scene):

```

```

    def construct(self):
        sq = Square(color=RED, fill_opacity=1)
        sq.move_to([-2, 0, 0])
        cr = Circle(color=BLUE, fill_opacity=1)
        cr.move_to([-1.3, 0.7, 0])
        un = Union(sq, cr, color=GREEN, fill_opacity=1)
        un.move_to([1.5, 0.3, 0])
        self.add(sq, cr, un)

"""

def __init__(self, *vmobjects: VMobject, **kwargs) -> None:
    if len(vmobjects) < 2:
        raise ValueError("At least 2 mobjects needed for Union.")
    super().__init__(**kwargs)
    paths = []
    for vmobject in vmobjects:
        paths.append(self._convert_vmobject_to_skia_path(vmobject))
    outpen = SkiaPath()
    union(paths, outpen.getPen())
    self._convert_skia_path_to_vmobject(outpen)

class Difference(_BooleanOps):
    """Subtracts one :class:`~.VMobject` from another one.

    Parameters
    -----
    subject
        The 1st :class:`~.VMobject`.
    clip
        The 2nd :class:`~.VMobject`


    Example
    -----
    .. manim:: DifferenceExample
        :save_last_frame:

        class DifferenceExample(Scene):
            def construct(self):
                sq = Square(color=RED, fill_opacity=1)
                sq.move_to([-2, 0, 0])
                cr = Circle(color=BLUE, fill_opacity=1)
                cr.move_to([-1.3, 0.7, 0])
                un = Difference(sq, cr, color=GREEN, fill_opacity=1)
                un.move_to([1.5, 0, 0])
                self.add(sq, cr, un)

    """

    def __init__(self, subject, clip, **kwargs) -> None:
        super().__init__(**kwargs)
        outpen = SkiaPath()
        difference(
            [self._convert_vmobject_to_skia_path(subject)],
            [self._convert_vmobject_to_skia_path(clip)],
            outpen.getPen(),
        )
        self._convert_skia_path_to_vmobject(outpen)

class Intersection(_BooleanOps):
    """Find the intersection of two :class:`~.VMobject`'s.
    This keeps the parts covered by both :class:`~.VMobject`'s.

    Parameters
    -----
    vmobjects
        The :class:`~.VMobject` to find the intersection.

    Raises
    -----
    ValueError
        If less than 2 :class:`~.VMobject` are passed.

    Example
    -----
    .. manim:: IntersectionExample
        :save_last_frame:

        class IntersectionExample(Scene):
            def construct(self):
                sq = Square(color=RED, fill_opacity=1)
                sq.move_to([-2, 0, 0])
                cr = Circle(color=BLUE, fill_opacity=1)
                cr.move_to([-1.3, 0.7, 0])
                un = Intersection(sq, cr, color=GREEN, fill_opacity=1)

```

```

        un.move_to([1.5, 0, 0])
        self.add(sq, cr, un)

"""

def __init__(self, *vmobjects, **kwargs) -> None:
    if len(vmobjects) < 2:
        raise ValueError("At least 2 mobjects needed for Intersection.")

    super().__init__(**kwargs)
    outpen = SkiaPath()
    intersection(
        [self._convert_vmobject_to_skia_path(vmobjects[0])],
        [self._convert_vmobject_to_skia_path(vmobjects[1])],
        outpen.getPen(),
    )
    new_outpen = outpen
    for _i in range(2, len(vmobjects)):
        new_outpen = SkiaPath()
        intersection(
            [outpen],
            [self._convert_vmobject_to_skia_path(vmobjects[_i])],
            new_outpen.getPen(),
        )
    outpen = new_outpen

    self._convert_skia_path_to_vmobject(outpen)

class Exclusion(_BooleanOps):
    """Find the XOR between two :class:`~.VMobject`.
    This creates a new :class:`~.VMobject` consisting of the region
    covered by exactly one of them.

    Parameters
    -----
    subject
        The 1st :class:`~.VMobject`.
    clip
        The 2nd :class:`~.VMobject`


    Example
    -----
    .. manim:: IntersectionExample
       :save_last_frame:

    class IntersectionExample(Scene):
        def construct(self):
            sq = Square(color=RED, fill_opacity=1)
            sq.move_to([-2, 0, 0])
            cr = Circle(color=BLUE, fill_opacity=1)
            cr.move_to([-1.3, 0.7, 0])
            un = Exclusion(sq, cr, color=GREEN, fill_opacity=1)
            un.move_to([1.5, 0.4, 0])
            self.add(sq, cr, un)

    """

    def __init__(self, subject, clip, **kwargs) -> None:
        super().__init__(**kwargs)
        outpen = SkiaPath()
        xor(
            [self._convert_vmobject_to_skia_path(subject)],
            [self._convert_vmobject_to_skia_path(clip)],
            outpen.getPen(),
        )
        self._convert_skia_path_to_vmobject(outpen)

```

group.py

```
"""Manim's cfg subcommand.

Manim's cfg subcommand is accessed in the command-line interface via ``manim
cfg``. Here you can specify options, subcommands, and subgroups for the cfg
group.

"""
from __future__ import annotations

import os
from ast import literal_eval
from pathlib import Path

import click
import cloup
from rich.errors import StyleSyntaxError
from rich.style import Style

from ... import cli_ctx_settings, console
from ..._config.utils import config_file_paths, make_config_parser
from ...constants import EPILOG
from ...utils.file_ops import guarantee_existence, open_file

RICH_COLOUR_INSTRUCTIONS: str = """
[red]The default colour is used by the input statement.
If left empty, the default colour will be used.[/red]
[magenta] For a full list of styles, visit[/magenta] [green]https://rich.readthedocs.io/en/latest/style.html[/green]
"""

RICH_NON_STYLE_ENTRIES: str = ["log.width", "log.height", "log.timestamps"]

def value_from_string(value: str) -> str | int | bool:
    """Extracts the literal of proper datatype from a string.

    Parameters
    -----
    value
        The value to check get the literal from.

    Returns
    -----
    Union[:class:`str`, :class:`int`, :class:`bool`]
        Returns the literal of appropriate datatype.
    """
    try:
        value = literal_eval(value)
    except (SyntaxError, ValueError):
        pass
    return value

def _is_expected_datatype(value: str, expected: str, style: bool = False) -> bool:
    """Checks whether `value` is the same datatype as `expected`,
    and checks if it is a valid `style` if `style` is true.

    Parameters
    -----
    value
        The string of the value to check (obtained from reading the user input).
    expected
        The string of the literal datatype must be matched by `value`. Obtained from
        reading the cfg file.
    style
        Whether or not to confirm if `value` is a style, by default False

    Returns
    -----
    :class:`bool`
        Whether or not `value` matches the datatype of `expected`.
    """
    value = value_from_string(value)
    expected = type(value_from_string(expected))

    return isinstance(value, expected) and (is_valid_style(value) if style else True)

def is_valid_style(style: str) -> bool:
    """Checks whether the entered color is a valid color according to rich

    Parameters
    -----
    style
        The style to check whether it is valid.

    Returns
    -----
    Boolean
        Returns whether it is valid style or not according to rich.
    """
    try:
        Style.parse(style)
        return True
    except StyleSyntaxError:
        return False
```

```

def replace_keys(default: dict) -> dict:
    """Replaces _ to . and vice versa in a dictionary for rich
    Parameters
    -----
    default
        The dictionary to check and replace
    Returns
    -----
    :class:`dict`
        The dictionary which is modified by replacing _ with . and vice versa
    """
    for key in default:
        if "_" in key:
            temp = default[key]
            del default[key]
            key = key.replace("_", ".")
            default[key] = temp
        else:
            temp = default[key]
            del default[key]
            key = key.replace(".", "_")
            default[key] = temp
    return default

@cloup.group(
    context_settings=cli_ctx_settings,
    invoke_without_command=True,
    no_args_is_help=True,
    epilog=EPILOG,
    help="Manages Manim configuration files.",
)
@click.pass_context
def cfg(ctx):
    """Responsible for the cfg subcommand."""
    pass

@cfg.command(context_settings=cli_ctx_settings, no_args_is_help=True)
@click.option(
    "-l",
    "--level",
    type=click.Choice(["user", "cwd"], case_sensitive=False),
    default="cwd",
    help="Specify if this config is for user or the working directory.",
)
@click.option("-o", "--open", "openfile", is_flag=True)
def write(level: str = None, openfile: bool = False) -> None:
    config_paths = config_file_paths()
    console.print(
        "[yellow bold]Manim Configuration File Writer[/yellow bold]",
        justify="center",
    )
    USER_CONFIG_MSG = f"""A configuration file at [yellow]{config_paths[1]}[/yellow] has been created with your required changes.
This will be used when running the manim command. If you want to override this config,
you will have to create a manim.cfg in the local directory, where you want those changes to be overridden."""
    CWD_CONFIG_MSG = f"""A configuration file at [yellow]{config_paths[2]}[/yellow] has been created.
To save your config please save that file and place it in your current working directory, from where you run the manim command."""

    parser = make_config_parser()
    if not openfile:
        action = "save this as"
        for category in parser:
            console.print(f'{category}', style="bold green underline")
            default = parser[category]
            if category == "logger":
                console.print(RICH_COLOUR_INSTRUCTIONS)
                default = replace_keys(default)

            for key in default:
                # All the cfg entries for logger need to be validated as styles,
                # as long as they aren't setting the log width or height etc
                if category == "logger" and key not in RICH_NON_STYLE_ENTRIES:
                    desc = "style"
                    style = default[key]
                else:
                    desc = "value"
                    style = None

                console.print(f"Enter the {desc} for {key} ", style=style, end="")
                if category != "logger" or key in RICH_NON_STYLE_ENTRIES:
                    defaultval = (
                        repr(default[key])
                        if isinstance(value_from_string(default[key]), str)
                        else default[key]
                    )
                    console.print(f"(defaults to {defaultval}) :", end="")
                try:
                    temp = input()
                except EOFError:
                    raise Exception(
                        """Not enough values in input.
You may have added a new entry to default.cfg, in which case you will have to
modify write_cfg_subcmd_input to account for it."""
                    )
                if temp:
                    while temp and not _is_expected_datatype(
                        temp,
                        style,
                    ):
                        temp = input("Enter the value for " + key + ": ")
                    if temp:
                        if category == "logger" and key == "log_level":
                            if temp not in ["INFO", "WARNING", "ERROR", "CRITICAL"]:
                                raise ValueError(
                                    f"Invalid log level '{temp}'. Valid levels are: INFO, WARNING, ERROR, CRITICAL."
                                )
                        if category == "logger" and key == "log_file":
                            if temp[-1] != "/":
                                temp += "/"
                        if category == "logger" and key == "log_directory":
                            if temp[-1] != "/":
                                temp += "/"
                        if category == "logger" and key == "log_color":
                            if temp not in ["true", "false"]:
                                raise ValueError(
                                    f"Invalid log color '{temp}'. Valid values are: true, false."
                                )
                        if category == "logger" and key == "log_max_files":
                            if not temp.isdigit():
                                raise ValueError(
                                    f"Invalid log max files '{temp}'. Value must be a positive integer."
                                )
                        if category == "logger" and key == "log_max_size":
                            if not temp.isdigit():
                                raise ValueError(
                                    f"Invalid log max size '{temp}'. Value must be a positive integer." +
                                    " Note: This value is in bytes, not kilobytes or megabytes."
                                )
                        if category == "logger" and key == "log_rotate_size":
                            if not temp.isdigit():
                                raise ValueError(
                                    f"Invalid log rotate size '{temp}'. Value must be a positive integer." +
                                    " Note: This value is in bytes, not kilobytes or megabytes."
                                )
                        if category == "logger" and key == "log_file_max_size":
                            if not temp.isdigit():
                                raise ValueError(
                                    f"Invalid log file max size '{temp}'. Value must be a positive integer." +
                                    " Note: This value is in bytes, not kilobytes or megabytes."
                                )
                        if category == "logger" and key == "log_file_max_backups":
                            if not temp.isdigit():
                                raise ValueError(
                                    f"Invalid log file max backups '{temp}'. Value must be a positive integer." +
                                    " Note: This value is in bytes, not kilobytes or megabytes."
                                )
                        if category == "logger" and key == "log_file_encoding":
                            if temp not in ["utf-8", "utf8", "latin1", "latin-1", "iso-8859-1", "iso8859-1", "cp1252", "cp1251", "cp1250", "cp1253", "cp1254", "cp1256", "cp1257", "cp1250-1257", "cp1250-1257-1258", "cp1250-1257-1258-1259", "cp1250-1257-1258-1259-1251", "cp1250-1257-1258-1259-1251-1252", "cp1250-1257-1258-1259-1251-1252-1253", "cp1250-1257-1258-1259-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254", "cp1250-1257-1258-1259-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-1251-1252-1253-1254-1255-1256-1257-1258-1259-1250-125
```

```

        temp,
        default[key],
        bool(style),
    ):
        console.print(
            f"[red bold]Invalid {desc}. Try again.[/red bold]",
        )
        console.print(
            f"Enter the {desc} for {key}:",
            style=style,
            end="",
        )
        temp = input()

    default[key] = temp.replace("%", "%%")

default = replace_keys(default) if category == "logger" else default

parser[category] = {
    i: v.replace("%", "%%") for i, v in dict(default).items()
}

else:
    action = "open"

if level is None:
    console.print(
        f"Do you want to {action} the default config for this User?(y/n)[[n]]",
        style="dim purple",
        end="",
    )
    action_to_userpath = input()
else:
    action_to_userpath = ""

if action_to_userpath.lower() == "y" or level == "user":
    cfg_file_path = config_paths[1]
    guarantee_existence(config_paths[1].parents[0])
    console.print(USER_CONFIG_MSG)
else:
    cfg_file_path = config_paths[2]
    guarantee_existence(config_paths[2].parents[0])
    console.print(CWD_CONFIG_MSG)
with cfg_file_path.open("w") as fp:
    parser.write(fp)
if openfile:
    open_file(cfg_file_path)

@cfg.command(context_settings=cli_ctx_settings)
def show():
    parser = make_config_parser()
    rich_non_style_entries = [a.replace(".", "_") for a in RICH_NON_STYLE_ENTRIES]
    for category in parser:
        console.print(f"{category}", style="bold green underline")
        for entry in parser[category]:
            if category == "logger" and entry not in rich_non_style_entries:
                console.print(f"{entry} :", end="")
                console.print(
                    f" {parser[category][entry]}",
                    style=parser[category][entry],
                )
            else:
                console.print(f"{entry} : {parser[category][entry]}")
    console.print("\n")

@cfg.command(context_settings=cli_ctx_settings)
@click.option("-d", "--directory", default=Path.cwd())
@click.pass_context
def export(ctx, directory):
    directory_path = Path(directory)
    if directory_path.absolute == Path.cwd().absolute:
        console.print(
            """You are reading the config from the same directory you are exporting to.
This means that the exported config will overwrite the config for this directory.
Are you sure you want to continue? (y/n)""",
            style="red bold",
            end="",
        )
        proceed = input().lower() == "y"
    else:
        proceed = True
    if proceed:
        if not directory_path.is_dir():
            console.print(f"Creating folder: {directory}.", style="red bold")
            directory_path.mkdir(parents=True)

    ctx.invoke(write)
    from_path = Path.cwd() / "manim.cfg"
    to_path = directory_path / "manim.cfg"

    console.print(f"Exported final Config at {from_path} to {to_path}.")
else:
    console.print("Aborted...", style="red bold")

```

shape_matchers.py

```
"""Mobjects used to mark and annotate other mobjects."""
from __future__ import annotations

__all__ = ["SurroundingRectangle", "BackgroundRectangle", "Cross", "Underline"]

from manim import config, logger
from manim.constants import *
from manim.mobject.geometry.line import Line
from manim.mobject.geometry.polygram import RoundedRectangle
from manim.mobject.mobject import Mobject
from manim.mobject.types.vectorized_mobject import VGroup
from manim.utils.color import BLACK, RED, YELLOW, Color, Colors

class SurroundingRectangle(RoundedRectangle):
    """A rectangle surrounding a :class:`~.Mobject`"""

    Examples
    -----
    .. manim:: SurroundingRectExample
        :save_last_frame:

    class SurroundingRectExample(Scene):
        def construct(self):
            title = Title("A Quote from Newton")
            quote = Text(
                "If I have seen further than others, \n"
                "it is by standing upon the shoulders of giants.",
                color=BLUE,
            ).scale(0.75)
            box = SurroundingRectangle(quote, color=YELLOW, buff=MED_LARGE_BUFF)

            t2 = Tex(r"Hello World").scale(1.5)
            box2 = SurroundingRectangle(t2, corner_radius=0.2)
            mobjects = VGroup(VGroup(box, quote), VGroup(t2, box2)).arrange(DOWN)
            self.add(title, mobjects)
    """

    def __init__(self, mobject, color=YELLOW, buff=SMALL_BUFF, corner_radius=0.0, **kwargs):
        super().__init__(
            color=color,
            width=mobject.width + 2 * buff,
            height=mobject.height + 2 * buff,
            corner_radius=corner_radius,
            **kwargs,
        )
        self.buff = buff
        self.move_to(mobject)

class BackgroundRectangle(SurroundingRectangle):
    """A background rectangle. Its default color is the background color of the scene."""

    Examples
    -----
    .. manim:: ExampleBackgroundRectangle
        :save_last_frame:

    class ExampleBackgroundRectangle(Scene):
        def construct(self):
            circle = Circle().shift(LEFT)
            circle.set_stroke(color=GREEN, width=20)
            triangle = Triangle().shift(2 * RIGHT)
            triangle.set_fill(PINK, opacity=0.5)
            backgroundRectangle1 = BackgroundRectangle(circle, color=WHITE, fill_opacity=0.15)
            backgroundRectangle2 = BackgroundRectangle(triangle, color=WHITE, fill_opacity=0.15)
            self.add(backgroundRectangle1)
            self.add(backgroundRectangle2)
            self.add(circle)
            self.add(triangle)
            self.play(Rotate(backgroundRectangle1, PI / 4))
            self.play(Rotate(backgroundRectangle2, PI / 2))
    """

    def __init__(self, mobject, color=YELLOW, buff=SMALL_BUFF, corner_radius=0.0, **kwargs):
        super().__init__(
            color=color,
            width=mobject.width + 2 * buff,
            height=mobject.height + 2 * buff,
            corner_radius=corner_radius,
            **kwargs,
        )
        self.buff = buff
        self.move_to(mobject)
```

```

color: Colors | None = None,
stroke_width: float = 0,
stroke_opacity: float = 0,
fill_opacity: float = 0.75,
buff: float = 0,
**kwargs,
):
    if color is None:
        color = config.background_color

    super().__init__(
        mobject,
        color=color,
        stroke_width=stroke_width,
        stroke_opacity=stroke_opacity,
        fill_opacity=fill_opacity,
        buff=buff,
        **kwargs,
    )
    self.original_fill_opacity = self.fill_opacity

def pointwiseBecomePartial(self, mobject, a, b):
    self.set_fill(opacity=b * self.original_fill_opacity)
    return self

def set_style(self, fill_opacity, **kwargs):
    # Unchangeable style, except for fill_opacity
    # All other style arguments are ignored
    super().set_style(
        stroke_color=BLACK,
        stroke_width=0,
        fill_color=BLACK,
        fill_opacity=fill_opacity,
    )
    if len(kwargs) > 0:
        logger.info(
            "Argument %s is ignored in BackgroundRectangle.set_style.",
            kwargs,
        )
    return self

def get_fill_color(self):
    return Color(self.color)

class Cross(VGroup):
    """Creates a cross.

Parameters
-----
mobject
    The mobject linked to this instance. It fits the mobject when specified. Defaults to None.
stroke_color
    Specifies the color of the cross lines. Defaults to RED.
stroke_width
    Specifies the width of the cross lines. Defaults to 6.
scale_factor
    Scales the cross to the provided units. Defaults to 1.

Examples
-----
.. manim:: ExampleCross
:save_last_frame:

    class ExampleCross(Scene):
        def construct(self):
            cross = Cross()
            self.add(cross)
"""

    def __init__(self,
                 mobject: Mobject | None = None,
                 stroke_color: Color = RED,
                 stroke_width: float = 6,
                 scale_factor: float = 1,
                 **kwargs,
                 ):
        super().__init__(
            Line(UP + LEFT, DOWN + RIGHT), Line(UP + RIGHT, DOWN + LEFT), **kwargs
        )
        if mobject is not None:
            self.replace(mobject, stretch=True)
        self.scale(scale_factor)
        self.set_stroke(color=stroke_color, width=stroke_width)

```

```
class Underline(Line):
    """Creates an underline.

Examples
-----
.. manim:: UnderLine
    :save_last_frame:

class UnderLine(Scene):
    def construct(self):
        man = Tex("Manim") # Full Word
        ul = Underline(man) # Underlining the word
        self.add(man, ul)
"""

def __init__(self, mobject, buff=SMALL_BUFF, **kwargs):
    super().__init__(LEFT, RIGHT, buff=buff, **kwargs)
    self.match_width(mobject)
    self.next_to(mobject, DOWN, buff=selfbuff)
```

commands.py_2

```
"""Manim's default subcommand, render.

Manim's render subcommand is accessed in the command-line interface via
```manim```, but can be more explicitly accessed with ```manim render````. Here you
can specify options, and arguments for the render command.

"""

from __future__ import annotations

import json
import sys
from pathlib import Path

import click
import cloup
import requests

from ... import __version__, config, console, error_console, logger
from ...config import tempconfig
from ...constants import EPILOG, RendererType
from ...utils.module_ops import scene_classes_from_file
from .ease_of_access_options import ease_of_access_options
from .global_options import global_options
from .output_options import output_options
from .render_options import render_options

@cloup.command(
 context_settings=None,
 no_args_is_help=True,
 epilog=EPILOG,
)
@click.argument("file", type=Path, required=True)
@click.argument("scene_names", required=False, nargs=-1)
@global_options
@output_options
@render_options # type: ignore
@ease_of_access_options
def render(
 **args,
):
 """Render SCENE(S) from the input FILE.

 FILE is the file path of the script or a config file.

 SCENES is an optional list of scenes in the file.
 """

 if args["save_as_gif"]:
 logger.warning("--save_as_gif is deprecated, please use --format=gif instead!")
 args["format"] = "gif"

 if args["save_pngs"]:
 logger.warning("--save_pngs is deprecated, please use --format=png instead!")
 args["format"] = "png"

 if args["show_in_file_browser"]:
 logger.warning(
 "The short form of show_in_file_browser is deprecated and will be moved to support --format.",
)

 class ClickArgs:
 def __init__(self, args):
 for name in args:
 setattr(self, name, args[name])

 def __get_kwargs__(self):
 return list(self.__dict__.items())

 def __eq__(self, other):
 if not isinstance(other, ClickArgs):
 return NotImplemented
 return vars(self) == vars(other)

 def __contains__(self, key):
 return key in self.__dict__

 def __repr__(self):
 return str(self.__dict__)

 click_args = ClickArgs(args)
 if args["jupyter"]:
 return click_args

 config.digest_args(click_args)
 file = Path(config.input_file)
 if config.renderer == RendererType.OPengl:
 from manim.renderer.opengl_renderer import OpenGLRenderer

 try:
 renderer = OpenGLRenderer()
 keep_running = True
```

```

while keep_running:
 for SceneClass in scene_classes_from_file(file):
 with tempconfig({}):
 scene = SceneClass(renderer)
 rerun = scene.render()
 if rerun or config["write_all"]:
 renderer.num_plays = 0
 continue
 else:
 keep_running = False
 break
 if config["write_all"]:
 keep_running = False

except Exception:
 error_console.print_exception()
 sys.exit(1)
else:
 for SceneClass in scene_classes_from_file(file):
 try:
 with tempconfig({}):
 scene = SceneClass()
 scene.render()
 except Exception:
 error_console.print_exception()
 sys.exit(1)

if config.notify_outdated_version:
 manim_info_url = "https://pypi.org/pypi/manim/json"
 warn_prompt = "Cannot check if latest release of manim is installed"
 req_info = {}

 try:
 req_info = requests.get(manim_info_url, timeout=10)
 req_info.raise_for_status()

 stable = req_info.json()["info"]["version"]
 if stable != __version__:
 console.print(
 f"You are using manim version [red]{__version__}[/red], but version [green]{stable}[/green] is available.",
)
 console.print(
 "You should consider upgrading via [yellow]pip install -U manim[/yellow]",
)
 except requests.exceptions.HTTPError:
 logger.debug(f"HTTP Error: {warn_prompt}")
 except requests.exceptions.ConnectionError:
 logger.debug(f"Connection Error: {warn_prompt}")
 except requests.exceptions.Timeout:
 logger.debug(f"Timed Out: {warn_prompt}")
 except json.JSONDecodeError:
 logger.debug(warn_prompt)
 logger.debug(f"Error decoding JSON from {manim_info_url}")
 except Exception:
 logger.debug(f"Something went wrong: {warn_prompt}")

return args

```

## commands.py\_3

```
from __future__ import annotations

import json
import os
from pathlib import Path
from subprocess import run
from typing import Generator

__all__ = [
 "capture",
 "get_video_metadata",
 "get_dir_layout",
]
]

def capture(command, cwd=None, command_input=None):
 p = run(command, cwd=cwd, input=command_input, capture_output=True, text=True)
 out, err = p.stdout, p.stderr
 return out, err, p.returncode

def get_video_metadata(path_to_video: str | os.PathLike) -> dict[str]:
 command = [
 "ffprobe",
 "-v",
 "error",
 "-select_streams",
 "v:0",
 "-show_entries",
 "stream=width,height,nb_frames,duration,avg_frame_rate,codec_name",
 "-print_format",
 "json",
 str(path_to_video),
]
 config, err, exitcode = capture(command)
 assert exitcode == 0, f"FFprobe error: {err}"
 return json.loads(config)["streams"][0]

def get_dir_layout(dirpath: Path) -> Generator[str, None, None]:
 """Get list of paths relative to dirpath of all files in dir and subdirs recursively."""
 for p in dirpath.iterdir():
 if p.is_dir():
 yield from get_dir_layout(p)
 continue
 yield str(p.relative_to(dirpath))
```

# moving\_camera\_scene.py

```
"""A scene whose camera can be moved around.

.. SEEALSO::

 :mod:`.moving_camera`

Examples

.. manim:: ChangingCameraWidthAndRestore

 class ChangingCameraWidthAndRestore(MovingCameraScene):
 def construct(self):
 text = Text("Hello World").set_color(BLUE)
 self.add(text)
 self.camera.frame.save_state()
 self.play(self.camera.frame.animate.set(width=text.width * 1.2))
 self.wait(0.3)
 self.play(Restore(self.camera.frame))

.. manim:: MovingCameraCenter

 class MovingCameraCenter(MovingCameraScene):
 def construct(self):
 s = Square(color=RED, fill_opacity=0.5).move_to(2 * LEFT)
 t = Triangle(color=GREEN, fill_opacity=0.5).move_to(2 * RIGHT)
 self.wait(0.3)
 self.add(s, t)
 self.play(self.camera.frame.animate.move_to(s))
 self.wait(0.3)
 self.play(self.camera.frame.animate.move_to(t))

.. manim:: MovingAndZoomingCamera

 class MovingAndZoomingCamera(MovingCameraScene):
 def construct(self):
 s = Square(color=BLUE, fill_opacity=0.5).move_to(2 * LEFT)
 t = Triangle(color=YELLOW, fill_opacity=0.5).move_to(2 * RIGHT)
 self.add(s, t)
 self.play(self.camera.frame.animate.move_to(s).set(width=s.width*2))
 self.wait(0.3)
 self.play(self.camera.frame.animate.move_to(t).set(width=t.width*2))

 self.play(self.camera.frame.animate.move_to(ORIGIN).set(width=14))

.. manim:: MovingCameraOnGraph

 class MovingCameraOnGraph(MovingCameraScene):
 def construct(self):
 self.camera.frame.save_state()

 ax = Axes(x_range=[-1, 10], y_range=[-1, 10])
 graph = ax.plot(lambda x: np.sin(x), color=WHITE, x_range=[0, 3 * PI])

 dot_1 = Dot(ax.i2gp(graph.t_min, graph))
 dot_2 = Dot(ax.i2gp(graph.t_max, graph))
 self.add(ax, graph, dot_1, dot_2)

 self.play(self.camera.frame.animate.scale(0.5).move_to(dot_1))
 self.play(self.camera.frame.animate.move_to(dot_2))
 self.play(Restore(self.camera.frame))
 self.wait()

"""

from __future__ import annotations

__all__ = ["MovingCameraScene"]

from manim.animation.animation import Animation

from ..camera.moving_camera import MovingCamera
from ..scene.scene import Scene
from ..utils.family import extract_mobject_family_members
from ..utils.iterables import list_update

class MovingCameraScene(Scene):
```

```
"""
This is a Scene, with special configurations and properties that
make it suitable for cases where the camera must be moved around.

.. SEEALSO::

 :class:`~.MovingCamera`

"""
def __init__(self, camera_class=MovingCamera, **kwargs):
 super().__init__(camera_class=camera_class, **kwargs)

def get_moving_mobjects(self, *animations: Animation):
 """
 This method returns a list of all of the Mobjects in the Scene that
 are moving, that are also in the animations passed.

 Parameters

 *animations
 The Animations whose mobjects will be checked.
 """
 moving_mobjects = super().get_moving_mobjects(*animations)
 all_moving_mobjects = extract_mobject_family_members(moving_mobjects)
 movement_indicators = self.renderer.camera.get_mobjects_indicating_movement()
 for movement_indicator in movement_indicators:
 if movement_indicator in all_moving_mobjects:
 # When one of these is moving, the camera should
 # consider all mobjects to be moving
 return list_update(self.mobjects, moving_mobjects)
 return moving_mobjects
```

# mobject.py

```
"""Base classes for objects that can be displayed."""
from __future__ import annotations

__all__ = ["Mobject", "Group", "override_animate"]

import copy
import itertools as it
import operator as op
import random
import sys
import types
import warnings
from functools import partialmethod, reduce
from math import ceil
from pathlib import Path
from typing import (
 TYPE_CHECKING,
 Callable,
 Dict,
 Iterable,
 List,
 Optional,
 Sequence,
 Tuple,
 Type,
 TypeVar,
 Union,
)
import numpy as np
from colour import Color

from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL

from .. import config, logger
from ..constants import *
from ..utils.color import (
 BLACK,
 WHITE,
 YELLOW_C,
 Colors,
 color_gradient,
 interpolate_color,
)
from ..utils.exceptions import MultiAnimationOverrideException
from ..utils.iterables import list_update, remove_list_redundancies
from ..utils.paths import straight_path
from ..utils.simple_functions import get_parameters
from ..utils.space_ops import angle_between_vectors, normalize, rotation_matrix

TODO: Explain array_attrs

Updater = Union[Callable[["Mobject"], None], Callable[["Mobject", float], None]]
T = TypeVar("T", bound="Mobject")

if TYPE_CHECKING:
 from ..animation.animation import Animation

class Mobject:
 """Mathematical Object: base class for objects that can be displayed on screen.

 There is a compatibility layer that allows for
 getting and setting generic attributes with ``get_*``
 and ``set_*`` methods. See :meth:`set` for more details.

 Attributes

 submobjects : List[:class:`~.Mobject`]
 The contained objects.
 points : :class:`numpy.ndarray`
 The points of the objects.

 .. seealso::

 :class:`~.VMobject`

 """

 animation_overrides = {}

 @classmethod
 def __init_subclass__(cls, **kwargs):
 super().__init_subclass__(**kwargs)

 cls.animation_overrides: dict[
 type[Animation],
 Callable[[Mobject], Animation],
] = {}
 cls._add_intrinsic_animation_overrides()
 cls._original_init__ = cls.__init__
```

```

def __init__(self, color=WHITE, name=None, dim=3, target=None, z_index=0):
 self.name = self.__class__.__name__ if name is None else name
 self.dim = dim
 self.target = target
 self.z_index = z_index
 self.point_hash = None
 self.subobjects = []
 self.updaters = []
 self.updating_suspended = False
 self.color = Color(color) if color else None

 self.reset_points()
 self.generate_points()
 self.init_colors()

@classmethod
def animation_override_for(
 cls,
 animation_class: type[Animation],
) -> Callable[[Mobject, ...], Animation] | None:
 """Returns the function defining a specific animation override for this class.

 Parameters

 animation_class
 The animation class for which the override function should be returned.

 Returns

 Optional[Callable[[Mobject, ...], Animation]]
 The function returning the override animation or ``None`` if no such animation
 override is defined.
 """
 if animation_class in cls.animation_overrides:
 return cls.animation_overrides[animation_class]

 return None

@classmethod
def _add_intrinsic_animation_overrides(cls):
 """Initializes animation overrides marked with the :func:`~.override_animation` decorator.

 """
 for method_name in dir(cls):
 # Ignore dunder methods
 if method_name.startswith("__"):
 continue

 method = getattr(cls, method_name)
 if hasattr(method, "_override_animation"):
 animation_class = method._override_animation
 cls.add_animation_override(animation_class, method)

@classmethod
def add_animation_override(
 cls,
 animation_class: type[Animation],
 override_func: Callable[[Mobject, ...], Animation],
):
 """Add an animation override.

 This does not apply to subclasses.

 Parameters

 animation_class
 The animation type to be overridden
 override_func
 The function returning an animation replacing the default animation. It gets
 passed the parameters given to the animation constructor.

 Raises

 MultiAnimationOverrideException
 If the overridden animation was already overridden.
 """
 if animation_class not in cls.animation_overrides:
 cls.animation_overrides[animation_class] = override_func
 else:
 raise MultiAnimationOverrideException(
 f"The animation {animation_class.__name__} for "
 f"{cls.__name__} is overridden by more than one method: "
 f"{cls.animation_overrides[animation_class].__qualname__} and "
 f"{override_func.__qualname__}.",
)

@classmethod
def set_default(cls, **kwargs):
 """Sets the default values of keyword arguments.

 If this method is called without any additional keyword
 arguments, the original default values of the initialization
 method of this class are restored.

 Parameters

 kwargs
 Passing any keyword argument will update the default

```

```
values of the keyword arguments of the initialization
function of this class.
```

#### Examples

-----

::

```
>>> from manim import Square, GREEN
>>> Square.set_default(color=GREEN, fill_opacity=0.25)
>>> s = Square(); s.color, s.fill_opacity
(<Color #83c167>, 0.25)
>>> Square.set_default()
>>> s = Square(); s.color, s.fill_opacity
(<Color white>, 0.0)

.. manim:: ChangedDefaultTextColor
:save_last_frame:

config.background_color = WHITE
```

```
class ChangedDefaultTextColor(Scene):
 def construct(self):
 Text.set_default(color=BLACK)
 self.add(Text("Changing default values is easy!"))

 # we revert the colour back to the default to prevent a bug in the docs.
 Text.set_default(color=WHITE)
```

"""

```
if kwargs:
 cls.__init__ = partialmethod(cls.__init__, **kwargs)
else:
 cls.__init__ = cls._original_init_
```

```
@property
def animate(self) -> _AnimationBuilder | T:
 """Used to animate the application of any method of :code:`self`.
```

Any method called on :code:`animate` is converted to an animation of applying that method on the mobject itself.

For example, :code:`square.set\_fill(WHITE)` sets the fill color of a square, while :code:`square.animate.set\_fill(WHITE)` animates this action.

Multiple methods can be put in a single animation once via chaining:

::

```
self.play(my_mobject.animate.shift(RIGHT).rotate(PI))
```

.. warning::

Passing multiple animations for the same :class:`Mobject` in one call to :meth:`~.Scene.play` is discouraged and will most likely not work properly. Instead of writing an animation like

::

```
self.play(my_mobject.animate.shift(RIGHT), my_mobject.animate.rotate(PI))
```

make use of method chaining.

Keyword arguments that can be passed to :meth:`.Scene.play` can be passed directly after accessing ```.animate```, like so::

```
self.play(my_mobject.animate(rate_func=linear).shift(RIGHT))
```

This is especially useful when animating simultaneous ```.animate```` calls that you want to behave differently::

```
self.play(
 mobject1.animate(run_time=2).rotate(PI),
 mobject2.animate(rate_func=there_and_back).shift(RIGHT),
)
```

.. seealso::

```
:func:`override_animate`
```

#### Examples

-----

.. manim:: AnimateExample

```
class AnimateExample(Scene):
 def construct(self):
 s = Square()
 self.play(Create(s))
 self.play(s.animate.shift(RIGHT))
 self.play(s.animate.scale(2))
 self.play(s.animate.rotate(PI / 2))
 self.play(Uncreate(s))
```

.. manim:: AnimateChainExample

```
class AnimateChainExample(Scene):
```

```

def construct(self):
 s = Square()
 self.play(Create(s))
 self.play(s.animate.shift(RIGHT).scale(2).rotate(PI / 2))
 self.play(Uncreate(s))

.. manim:: AnimateWithArgsExample

class AnimateWithArgsExample(Scene):
 def construct(self):
 s = Square()
 c = Circle()

 VGroup(s, c).arrange(RIGHT, buff=2)
 self.add(s, c)

 self.play(
 s.animate(run_time=2).rotate(PI / 2),
 c.animate(rate_func=there_and_back).shift(RIGHT),
)

.. warning::

    ```.animate```
    will interpolate the :class:`~.Mobject` between its points prior to
    ```.animate``` and its points after applying ```.animate``` to it. This may
 result in unexpected behavior when attempting to interpolate along paths,
 or rotations.
 If you want animations to consider the points between, consider using
 :class:`~.ValueTracker` with updaters instead.

"""
return _AnimationBuilder(self)

def __deepcopy__(self, clone_from_id):
 cls = self.__class__
 result = cls.__new__(cls)
 clone_from_id[id(self)] = result
 for k, v in self.__dict__.items():
 setattr(result, k, copy.deepcopy(v, clone_from_id))
 result.original_id = str(id(self))
 return result

def __repr__(self):
 return str(self.name)

def reset_points(self):
 """Sets :attr:`points` to be an empty array."""
 self.points = np.zeros((0, self.dim))

def init_colors(self):
 """Initializes the colors.

 Gets called upon creation. This is an empty method that can be implemented by
 subclasses.
 """
 pass

def generate_points(self):
 """Initializes :attr:`points` and therefore the shape.

 Gets called upon creation. This is an empty method that can be implemented by
 subclasses.
 """
 pass

def add(self, *mobjects: Mobject):
 """Add mobjects as submobjects.

 The mobjects are added to :attr:`submobjects`.

 Subclasses of mobject may implement ``+`` and ``+=`` dunder methods.

 Parameters

 mobjects
 The mobjects to add.

 Returns

 :class:`Mobject`
 ``self``

 Raises

 :class:`ValueError`
 When a mobject tries to add itself.
 :class:`TypeError`
 When trying to add an object that is not an instance of :class:`Mobject`.

 Notes

 A mobject cannot contain itself, and it cannot contain a submobject
 more than once. If the parent mobject is displayed, the newly-added
 submobjects will also be displayed (i.e. they are automatically added
 to the parent Scene).
 """

See Also
```

```

:meth:`remove`
:meth:`add_to_back`

Examples

::

>>> outer = Mobject()
>>> inner = Mobject()
>>> outer = outer.add(inner)

Duplicates are not added again::

>>> outer = outer.add(inner)
>>> len(outer.submobjects)
1

Adding an object to itself raises an error::

>>> outer.add(outer)
Traceback (most recent call last):
...
ValueError: Mobject cannot contain self

A given mobject cannot be added as a submobject
twice to some parent::

>>> parent = Mobject(name="parent")
>>> child = Mobject(name="child")
>>> parent.add(child, child)
[...] WARNING ...
parent
>>> parent.submobjects
[child]

"""
for m in mobjects:
 if not isinstance(m, Mobject):
 raise TypeError("All submobjects must be of type Mobject")
 if m is self:
 raise ValueError("Mobject cannot contain self")

unique_mobjects = remove_list_redundancies(mobjects)
if len(mobjects) != len(unique_mobjects):
 logger.warning(
 "Attempted adding some Mobject as a child more than once, "
 "this is not possible. Repetitions are ignored.",
)
self.submobjects = list_update(self.submobjects, unique_mobjects)
return self

def insert(self, index: int, mobject: Mobject):
 """Inserts a mobject at a specific position into self.submobjects

Effectively just calls ``self.submobjects.insert(index, mobject)``,
where ``self.submobjects`` is a list.

Highly adapted from ``Mobject.add``.

Parameters

index
 The index at which
mobject
 The mobject to be inserted.
"""

if not isinstance(mobject, Mobject):
 raise TypeError("All submobjects must be of type Mobject")
if mobject is self:
 raise ValueError("Mobject cannot contain self")
self.submobjects.insert(index, mobject)

def __add__(self, mobject):
 raise NotImplementedError

def __iadd__(self, mobject):
 raise NotImplementedError

def add_to_back(self, *mobjects: Mobject):
 """Add all passed mobjects to the back of the submobjects.

If :attr:`submobjects` already contains the given mobjects, they just get moved
to the back instead.

Parameters

mobjects
 The mobjects to add.

Returns

:class:`Mobject`
 ``self``
.. note::
```

Technically, this is done by adding (or moving) the mobjects to the head of :attr:`submobjects`. The head of this list is rendered first, which places the corresponding mobjects behind the subsequent list members.

Raises

-----  
:class:`ValueError`  
    When a mobject tries to add itself.  
:class:`TypeError`  
    When trying to add an object that is not an instance of :class:`Mobject`.

Notes

-----  
A mobject cannot contain itself, and it cannot contain a submobject more than once. If the parent mobject is displayed, the newly-added submobjects will also be displayed (i.e. they are automatically added to the parent Scene).

See Also

-----  
:meth:`remove`  
:meth:`add`

"""  
if self in mobjects:  
    raise ValueError("A mobject shouldn't contain itself")  
  
for mobject in mobjects:  
    if not isinstance(mobject, Mobject):  
        raise TypeError("All submobjects must be of type Mobject")  
  
self.remove(\*mobjects)  
# dict.fromkeys() removes duplicates while maintaining order  
self.submobjects = list(dict.fromkeys(mobjects)) + self.submobjects  
return self

def remove(self, \*mobjects: Mobject):  
 """Remove :attr:`submobjects`.

The mobjects are removed from :attr:`submobjects`, if they exist.

Subclasses of mobject may implement ``-`` and ``-=`` dunder methods.

Parameters

-----  
mobjects  
    The mobjects to remove.

Returns

-----  
:class:`Mobject`  
    ``self``

See Also

-----  
:meth:`add`

"""  
for mobject in mobjects:  
    if mobject in self.submobjects:  
        self.submobjects.remove(mobject)  
return self

def \_\_sub\_\_(self, other):  
 raise NotImplementedError

def \_\_isub\_\_(self, other):  
 raise NotImplementedError

def set(self, \*\*kwargs):  
 """Sets attributes.

I.e. ``my\_mobject.set(foo=1)`` applies ``my\_mobject.foo = 1``.

This is a convenience to be used along with :attr:`animate` to animate setting attributes.

In addition to this method, there is a compatibility layer that allows ``get\_\*`` and ``set\_\*`` methods to get and set generic attributes. For instance::

```
>>> mob = Mobject()
>>> mob.set_foo(0)
Mobject
>>> mob.get_foo()
0
>>> mob.foo
0
```

This compatibility layer does not interfere with any ``get\_\*`` or ``set\_\*`` methods that are explicitly defined.

.. warning::

This compatibility layer is for backwards compatibility and is not guaranteed to stay around. Where applicable, please prefer getting/setting attributes normally or with

```

the :meth:`set` method.

Parameters

**kwargs
 The attributes and corresponding values to set.

Returns

:class:`~Mobject`
 ``self``

Examples

::

 >>> mob = Mobject()
 >>> mob.set(foo=0)
 Mobject
 >>> mob.foo
 0
"""

for attr, value in kwargs.items():
 setattr(self, attr, value)

return self

def __getattr__(self, attr):
 # Add automatic compatibility layer
 # between properties and get_* and set_*
 # methods.
 #
 # In python 3.9+ we could change this
 # logic to use str.remove_prefix instead.

 if attr.startswith("get_"):
 # Remove the "get_" prefix
 to_get = attr[4:]

 def getter(self):
 warnings.warn(
 "This method is not guaranteed to stay around. Please prefer "
 "getting the attribute normally.",
 DeprecationWarning,
 stacklevel=2,
)

 return getter

 # Return a bound method
 return types.MethodType(getter, self)

if attr.startswith("set_"):
 # Remove the "set_" prefix
 to_set = attr[4:]

 def setter(self, value):
 warnings.warn(
 "This method is not guaranteed to stay around. Please prefer "
 "setting the attribute normally or with Mobject.set().",
 DeprecationWarning,
 stacklevel=2,
)

 setattr(self, to_set, value)

 return setter

Return a bound method
return types.MethodType(setter, self)

Unhandled attribute, therefore error
raise AttributeError(f"{type(self).__name__} object has no attribute '{attr}'")

@property
def width(self):
 """The width of the mobject.

Returns

:class:`float`

Examples

.. manim:: WidthExample

 class WidthExample(Scene):
 def construct(self):
 decimal = DecimalNumber().to_edge(UP)
 rect = Rectangle(color=BLUE)
 rect_copy = rect.copy().set_stroke(GRAY, opacity=0.5)

 decimal.add_updater(lambda d: d.set_value(rect.width))

 self.add(rect_copy, rect, decimal)
 self.play(rect.animate.set(width=7))
 self.wait()

```

```

See also

:meth:`length_over_dim`

"""
Get the length across the X dimension
return self.length_over_dim(0)

@width.setter
def width(self, value):
 self.scale_to_fit_width(value)

@property
def height(self):
 """The height of the mobject.

 Returns

 :class:`float`

 Examples

 .. manim:: HeightExample

 class HeightExample(Scene):
 def construct(self):
 decimal = DecimalNumber().to_edge(UP)
 rect = Rectangle(color=BLUE)
 rect_copy = rect.copy().set_stroke(GRAY, opacity=0.5)

 decimal.add_updater(lambda d: d.set_value(rect.height))

 self.add(rect_copy, rect, decimal)
 self.play(rect.animate.set(height=5))
 self.wait()

 See also

 :meth:`length_over_dim`

"""
Get the length across the Y dimension
return self.length_over_dim(1)

@height.setter
def height(self, value):
 self.scale_to_fit_height(value)

@property
def depth(self):
 """The depth of the mobject.

 Returns

 :class:`float`

 See also

 :meth:`length_over_dim`

"""
Get the length across the Z dimension
return self.length_over_dim(2)

@depth.setter
def depth(self, value):
 self.scale_to_fit_depth(value)

def get_array_attrs(self):
 return ["points"]

def apply_over_attr_arrays(self, func):
 for attr in self.get_array_attrs():
 setattr(self, attr, func(getattr(self, attr)))
 return self

Displaying

def get_image(self, camera=None):
 if camera is None:
 from ..camera.camera import Camera

 camera = Camera()
 camera.capture_mobject(self)
 return camera.get_image()

def show(self, camera=None):
 self.get_image(camera=camera).show()

def save_image(self, name=None):
 """Saves an image of only this :class:`Mobject` at its position to a png file."""
 self.get_image().save(
 Path(config.get_dir("video_dir")).joinpath((name or str(self)) + ".png"),
)

```

```

def copy(self: T) -> T:
 """Create and return an identical copy of the :class:`Mobject` including all
 :attr:`submobjects`.

 Returns

 :class:`Mobject`
 The copy.

 Note

 The clone is initially not visible in the Scene, even if the original was.

 """
 return copy.deepcopy(self)

def generate_target(self, use_deepcopy=False):
 self.target = None # Prevent unbounded linear recursion
 if use_deepcopy:
 self.target = copy.deepcopy(self)
 else:
 self.target = self.copy()
 return self.target

Updating

def update(self, dt: float = 0, recursive: bool = True):
 """Apply all updaters.

 Does nothing if updating is suspended.

 Parameters

 dt
 The parameter ``dt`` to pass to the update functions. Usually this is the
 time in seconds since the last call of ``update``.
 recursive
 Whether to recursively update all submobjects.

 Returns

 :class:`Mobject`
 ``self``

 See Also

 :meth:`add_updater`
 :meth:`get_updaters`

 """
 if self.updating_suspended:
 return self
 for updater in self.updaters:
 parameters = get_parameters(updater)
 if "dt" in parameters:
 updater(self, dt)
 else:
 updater(self)
 if recursive:
 for submob in self.submobjects:
 submob.update(dt, recursive)
 return self

def get_time_based_updaters(self) -> list[Updater]:
 """Return all updaters using the ``dt`` parameter.

 The updaters use this parameter as the input for difference in time.

 Returns

 List[:class:`Callable`]
 The list of time based updaters.

 See Also

 :meth:`get_updaters`
 :meth:`has_time_based_updater`

 """
 return [updater for updater in self.updaters if "dt" in get_parameters(updater)]

def has_time_based_updater(self) -> bool:
 """Test if ``self`` has a time based updater.

 Returns

 class:`bool`
 ``True`` if at least one updater uses the ``dt`` parameter, ``False``
 otherwise.

 See Also

 :meth:`get_time_based_updaters`

 """
 return any("dt" in get_parameters(updater) for updater in self.updaters)

def get_updaters(self) -> list[Updater]:
 """Return all updaters.

```

```

>Returns

List[:class:`Callable`]
 The list of updaters.

See Also

:meth:`~.add_updater`
:meth:`~.get_time_based_updaters`

"""
return self.updaters

def get_family_updaters(self):
 return list(it.chain(*[sm.get_updaters() for sm in self.get_family()]))

def add_updater(
 self,
 update_function: Updater,
 index: int | None = None,
 call_updater: bool = False,
):
 """Add an update function to this mobject.

 Update functions, or updaters in short, are functions that are applied to the
 Mobject in every frame.

 Parameters

 update_function
 The update function to be added.
 Whenever :meth:`update` is called, this update function gets called using
 ``self`` as the first parameter.
 The updater can have a second parameter ``dt``. If it uses this parameter,
 it gets called using a second value ``dt``, usually representing the time
 in seconds since the last call of :meth:`update`.
 index
 The index at which the new updater should be added in ``self.updaters``.
 In case ``index`` is ``None`` the updater will be added at the end.
 call_updater
 Whether or not to call the updater initially. If ``True``, the updater will
 be called using ``dt=0``.

>Returns

:class:`Mobject`
 ``self``

Examples

.. manim:: NextToUpdateer

 class NextToUpdateer(Scene):
 def construct(self):
 def dot_position(mobject):
 mobject.set_value(dot.get_center()[0])
 mobject.next_to(dot)

 dot = Dot(RIGHT*3)
 label = DecimalNumber()
 label.add_updater(dot_position)
 self.add(dot, label)

 self.play(Rotating(dot, about_point=ORIGIN, angle=TAU, run_time=TAU, rate_func=linear))

.. manim:: DtUpdater

 class DtUpdater(Scene):
 def construct(self):
 line = Square()

 #Let the line rotate 90° per second
 line.add_updater(lambda mobject, dt: mobject.rotate(dt*90*DEGREES))
 self.add(line)
 self.wait(2)

See also

:meth:`~.get_updaters`
:meth:`~.remove_updater`
:class:`~.UpdateFromFunc`"""

if index is None:
 self.updaters.append(update_function)
else:
 self.updaters.insert(index, update_function)
if call_updater:
 parameters = get_parameters(update_function)
 if "dt" in parameters:
 update_function(self, 0)
 else:
 update_function(self)
return self

def remove_updater(self, update_function: Updater):
 """Remove an updater.

 If the same updater is applied multiple times, every instance gets removed.

```

```

Parameters

update_function
 The update function to be removed.

Returns

:class:`Mobject`

``self``

See also

:meth:`clear_updaters`

:meth:`add_updater`

:meth:`get_updaters`

"""

while update_function in self.updaters:

 self.updaters.remove(update_function)

return self

def clear_updaters(self, recursive: bool = True):
 """Remove every updater.

Parameters

recursive
 Whether to recursively call ``clear_updaters`` on all submobjects.

Returns

:class:`Mobject`

``self``

See also

:meth:`remove_updater`

:meth:`add_updater`

:meth:`get_updaters`

"""

self.updaters = []

if recursive:

 for submob in self.submobjects:

 submob.clear_updaters()

return self

def match_updaters(self, mobject: Mobject):
 """Match the updaters of the given mobject.

Parameters

mobject
 The mobject whose updaters get matched.

Returns

:class:`Mobject`

``self``

Note

All updaters from submobjects are removed, but only updaters of the given
mobject are matched, not those of it's submobjects.

See also

:meth:`add_updater`

:meth:`clear_updaters`

"""

self.clear_updaters()

for updater in mobject.get_updaters():
 self.add_updater(updater)
return self

def suspend_updating(self, recursive: bool = True):
 """Disable updating from updaters and animations.

Parameters

recursive
 Whether to recursively suspend updating on all submobjects.

Returns

:class:`Mobject`

``self``

See also

:meth:`resume_updating`

:meth:`add_updater`

"""

```

```

self.updating_suspended = True
if recursive:
 for submob in self.submobjects:
 submob.suspend_updating(recursive)
return self

def resume_updating(self, recursive: bool = True):
 """Enable updating from updaters and animations.

 Parameters

 recursive
 Whether to recursively enable updating on all submobjects.

 Returns

 :class:`Mobject`
 ``self``

 See also

 :meth:`suspend_updating`
 :meth:`add_updater`

 """
 self.updating_suspended = False
 if recursive:
 for submob in self.submobjects:
 submob.resume_updating(recursive)
 self.update(dt=0, recursive=recursive)
 return self

Transforming operations

def apply_to_family(self, func: Callable[[Mobject], None]):
 """Apply a function to ``self`` and every submobject with points recursively.

 Parameters

 func
 The function to apply to each mobject. ``func`` gets passed the respective
 (sub)mobject as parameter.

 Returns

 :class:`Mobject`
 ``self``

 See also

 :meth:`family_members_with_points`

 """
 for mob in self.family_members_with_points():
 func(mob)

def shift(self, *vectors: np.ndarray):
 """Shift by the given vectors.

 Parameters

 vectors
 Vectors to shift by. If multiple vectors are given, they are added
 together.

 Returns

 :class:`Mobject`
 ``self``

 See also

 :meth:`move_to`

 """
 total_vector = reduce(op.add, vectors)
 for mob in self.family_members_with_points():
 mob.points = mob.points.astype("float")
 mob.points += total_vector

 return self

def scale(self, scale_factor: float, **kwargs):
 r"""Scale the size by a factor.

 Default behavior is to scale about the center of the mobject.

 Parameters

 scale_factor
 The scaling factor :math:`|\alpha| < 1` , the mobject
 will shrink, and for :math:`|\alpha| > 1` it will grow. Furthermore,
 if :math:`\alpha < 0` , the mobject is also flipped.
 kwargs
 Additional keyword arguments passed to
 :meth:`apply_points_function_about_point`.

 Returns

```

```

:class:`Mobject`

` ``self```

Examples

.. manim:: MobjectScaleExample

:save_last_frame:

class MobjectScaleExample(Scene):

 def construct(self):

 f1 = Text("F")

 f2 = Text("F").scale(2)

 f3 = Text("F").scale(0.5)

 f4 = Text("F").scale(-1)

 vgroup = VGroup(f1, f2, f3, f4).arrange(6 * RIGHT)

 self.add(vgroup)

See also

:meth:`move_to`

"""
self.apply_points_function_about_point(

 lambda points: scale_factor * points, **kwargs

)
return self

def rotate_about_origin(self, angle, axis=OUT, axes=[]):

 """Rotates the :class:`Mobject` about the ORIGIN, which is at [0,0,0]."""

 return self.rotate(angle, axis, about_point=ORIGIN)

def rotate(

 self,

 angle,

 axis=OUT,

 about_point: Sequence[float] | None = None,

 **kwargs,

):

 """Rotates the :class:`Mobject` about a certain point."""

 rot_matrix = rotation_matrix(angle, axis)

 self.apply_points_function_about_point(

 lambda points: np.dot(points, rot_matrix.T), about_point, **kwargs
)
return self

def flip(self, axis=UP, **kwargs):

 """Flips/Mirrors an mobject about its center.

Examples

.. manim:: FlipExample

:save_last_frame:

class FlipExample(Scene):

 def construct(self):

 s= Line(LEFT, RIGHT+UP).shift(4*LEFT)

 self.add(s)

 s2= s.copy().flip()

 self.add(s2)

"""
return self.rotate(TAU / 2, axis, **kwargs)

def stretch(self, factor, dim, **kwargs):

 def func(points):

 points[:, dim] *= factor

 return points

 self.apply_points_function_about_point(func, **kwargs)

 return self

def apply_function(self, function, **kwargs):

 # Default to applying matrix about the origin, not mobjects center

 if len(kwargs) == 0:

 kwargs["about_point"] = ORIGIN

 self.apply_points_function_about_point(

 lambda points: np.apply_along_axis(function, 1, points), **kwargs
)
return self

def apply_function_to_position(self, function):

 self.move_to(function(self.get_center()))

 return self

def apply_function_to_submobject_positions(self, function):

 for submob in self.submobjects:

 submob.apply_function_to_position(function)

 return self

def apply_matrix(self, matrix, **kwargs):

 # Default to applying matrix about the origin, not mobjects center

 if ("about_point" not in kwargs) and ("about_edge" not in kwargs):

 kwargs["about_point"] = ORIGIN

 full_matrix = np.identity(self.dim)

 matrix = np.array(matrix)

```

```

full_matrix[: matrix.shape[0], : matrix.shape[1]] = matrix
self.apply_points_function_about_point(
 lambda points: np.dot(points, full_matrix.T), **kwargs
)
return self

def apply_complex_function(self, function, **kwargs):
 """Applies a complex function to a :class:`Mobject`.
 The x and y coordinates correspond to the real and imaginary parts respectively.
 """
 Example

 .. manim:: ApplyFuncExample

 class ApplyFuncExample(Scene):
 def construct(self):
 circ = Circle().scale(1.5)
 circ_ref = circ.copy()
 circ.apply_complex_function(
 lambda x: np.exp(x*1j)
)
 t = ValueTracker(0)
 circ.add_updater(
 lambda x: x.become(circ_ref.copy().apply_complex_function(
 lambda x: np.exp(x+t.get_value()*1j)
)).set_color(BLUE)
)
 self.add(circ_ref)
 self.play(TransformFromCopy(circ_ref, circ))
 self.play(t.animate.set_value(TAU), run_time=3)
 """
 def R3_func(point):
 x, y, z = point
 xy_complex = function(complex(x, y))
 return [xy_complex.real, xy_complex.imag, z]

 return self.apply_function(R3_func)

def wag(self, direction=RIGHT, axis=DOWN, wag_factor=1.0):
 for mob in self.family_members_with_points():
 alphas = np.dot(mob.points, np.transpose(axis))
 alphas -= min(alphas)
 alphas /= max(alphas)
 alphas = alphas**wag_factor
 mob.points += np.dot(
 alphas.reshape((len(alphas), 1)),
 np.array(direction).reshape((1, mob.dim)),
)
 return self

def reverse_points(self):
 for mob in self.family_members_with_points():
 mob.apply_over_attr_arrays(lambda arr: np.array(list(reversed(arr))))
 return self

def repeat(self, count: int):
 """This can make transition animations nicer"""

 def repeat_array(array):
 return reduce(lambda a1, a2: np.append(a1, a2, axis=0), [array] * count)

 for mob in self.family_members_with_points():
 mob.apply_over_attr_arrays(repeat_array)
 return self

In place operations.
Note, much of these are now redundant with default behavior of
above methods

def apply_points_function_about_point(
 self,
 func,
 about_point=None,
 about_edge=None,
):
 if about_point is None:
 if about_edge is None:
 about_edge = ORIGIN
 about_point = self.get_critical_point(about_edge)
 for mob in self.family_members_with_points():
 mob.points -= about_point
 mob.points = func(mob.points)
 mob.points += about_point
 return self

def pose_at_angle(self, **kwargs):
 self.rotate(TAU / 14, RIGHT + UP, **kwargs)
 return self

Positioning methods

def center(self):
 self.shift(-self.get_center())
 return self

def align_on_border(self, direction, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
 """Direction just needs to be a vector pointing towards side or
 """

```

```

corner in the 2d plane.
"""
target_point = np.sign(direction) * (
 config["frame_x_radius"],
 config["frame_y_radius"],
 0,
)
point_to_align = self.get_critical_point(direction)
shift_val = target_point - point_to_align - buff * np.array(direction)
shift_val = shift_val * abs(np.sign(direction))
self.shift(shift_val)
return self

def to_corner(self, corner=LEFT + DOWN, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
 return self.align_on_border(corner, buff)

def to_edge(self, edge=LEFT, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
 return self.align_on_border(edge, buff)

def next_to(
 self,
 mobject_or_point,
 direction=RIGHT,
 buff=DEFAULT_MOBJECT_TO_MOBJECT_BUFFER,
 aligned_edge=ORIGIN,
 submobject_to_align=None,
 index_of_submobject_to_align=None,
 coor_mask=np.array([1, 1, 1]),
):
 """Move this :class:`~.Mobject` next to another's :class:`~.Mobject` or coordinate.

Examples

.. manim:: GeometricShapes
:save_last_frame:

 class GeometricShapes(Scene):
 def construct(self):
 d = Dot()
 c = Circle()
 s = Square()
 t = Triangle()
 d.next_to(c, RIGHT)
 s.next_to(c, LEFT)
 t.next_to(c, DOWN)
 self.add(d, c, s, t)

 if isinstance(mobject_or_point, Mobject):
 mob = mobject_or_point
 if index_of_submobject_to_align is not None:
 target_aligner = mob[index_of_submobject_to_align]
 else:
 target_aligner = mob
 target_point = target_aligner.get_critical_point(aligned_edge + direction)
 else:
 target_point = mobject_or_point
 if submobject_to_align is not None:
 aligner = submobject_to_align
 elif index_of_submobject_to_align is not None:
 aligner = self[index_of_submobject_to_align]
 else:
 aligner = self
 point_to_align = aligner.get_critical_point(aligned_edge - direction)
 self.shift((target_point - point_to_align + buff * direction) * coor_mask)
 return self

def shift_onto_screen(self, **kwargs):
 space_lengths = [config["frame_x_radius"], config["frame_y_radius"]]
 for vect in UP, DOWN, LEFT, RIGHT:
 dim = np.argmax(np.abs(vect))
 buff = kwargs.get("buff", DEFAULT_MOBJECT_TO_EDGE_BUFFER)
 max_val = space_lengths[dim] - buff
 edge_center = self.get_edge_center(vect)
 if np.dot(edge_center, vect) > max_val:
 self.to_edge(vect, **kwargs)
 return self

def is_off_screen(self):
 if self.get_left()[0] > config["frame_x_radius"]:
 return True
 if self.get_right()[0] < -config["frame_x_radius"]:
 return True
 if self.get_bottom()[1] > config["frame_y_radius"]:
 return True
 if self.get_top()[1] < -config["frame_y_radius"]:
 return True
 return False

def stretch_about_point(self, factor, dim, point):
 return self.stretch(factor, dim, about_point=point)

def rescale_to_fit(self, length, dim, stretch=False, **kwargs):
 old_length = self.length_over_dim(dim)
 if old_length == 0:
 return self
 if stretch:
 self.stretch(length / old_length, dim, **kwargs)

```

```

 else:
 self.scale(length / old_length, **kwargs)
 return self

def scale_to_fit_width(self, width, **kwargs):
 """Scales the :class:`~.Mobject` to fit a width while keeping height/depth proportional.

 Returns

 :class:`Mobject`
 ``self``

 Examples

 ::

 >>> from manim import *
 >>> sq = Square()
 >>> sq.height
 2.0
 >>> sq.scale_to_fit_width(5)
 Square
 >>> sq.width
 5.0
 >>> sq.height
 5.0
 """

 return self.rescale_to_fit(width, 0, stretch=False, **kwargs)

def stretch_to_fit_width(self, width, **kwargs):
 """Stretches the :class:`~.Mobject` to fit a width, not keeping height/depth proportional.

 Returns

 :class:`Mobject`
 ``self``

 Examples

 ::

 >>> from manim import *
 >>> sq = Square()
 >>> sq.height
 2.0
 >>> sq.stretch_to_fit_width(5)
 Square
 >>> sq.width
 5.0
 >>> sq.height
 2.0
 """

 return self.rescale_to_fit(width, 0, stretch=True, **kwargs)

def scale_to_fit_height(self, height, **kwargs):
 """Scales the :class:`~.Mobject` to fit a height while keeping width/depth proportional.

 Returns

 :class:`Mobject`
 ``self``

 Examples

 ::

 >>> from manim import *
 >>> sq = Square()
 >>> sq.width
 2.0
 >>> sq.scale_to_fit_height(5)
 Square
 >>> sq.height
 5.0
 >>> sq.width
 5.0
 """

 return self.rescale_to_fit(height, 1, stretch=False, **kwargs)

def stretch_to_fit_height(self, height, **kwargs):
 """Stretches the :class:`~.Mobject` to fit a height, not keeping width/depth proportional.

 Returns

 :class:`Mobject`
 ``self``

 Examples

 ::

 >>> from manim import *
 >>> sq = Square()
 >>> sq.width
 2.0
 >>> sq.stretch_to_fit_height(5)

```

```

 Square
 >>> sq.height
 5.0
 >>> sq.width
 2.0
"""

 return self.rescale_to_fit(height, 1, stretch=True, **kwargs)

def scale_to_fit_depth(self, depth, **kwargs):
 """Scales the :class:`~.Mobject` to fit a depth while keeping width/height proportional."""
 return self.rescale_to_fit(depth, 2, stretch=False, **kwargs)

def stretch_to_fit_depth(self, depth, **kwargs):
 """Stretches the :class:`~.Mobject` to fit a depth, not keeping width/height proportional."""
 return self.rescale_to_fit(depth, 2, stretch=True, **kwargs)

def set_coord(self, value, dim, direction=ORIGIN):
 curr = self.get_coord(dim, direction)
 shift_vect = np.zeros(self.dim)
 shift_vect[dim] = value - curr
 self.shift(shift_vect)
 return self

def set_x(self, x, direction=ORIGIN):
 """Set x value of the center of the :class:`~.Mobject` (`int` or `float`).
 """
 return self.set_coord(x, 0, direction)

def set_y(self, y, direction=ORIGIN):
 """Set y value of the center of the :class:`~.Mobject` (`int` or `float`).
 """
 return self.set_coord(y, 1, direction)

def set_z(self, z, direction=ORIGIN):
 """Set z value of the center of the :class:`~.Mobject` (`int` or `float`).
 """
 return self.set_coord(z, 2, direction)

def space_out_submobjects(self, factor=1.5, **kwargs):
 self.scale(factor, **kwargs)
 for submob in self.submobjects:
 submob.scale(1.0 / factor)
 return self

def move_to(
 self,
 point_or_mobject,
 aligned_edge=ORIGIN,
 coor_mask=np.array([1, 1, 1]),
):
 """Move center of the :class:`~.Mobject` to certain coordinate.
 If `point_or_mobject` is an instance of :class:`Mobject`, target is its critical point along the aligned edge.
 Otherwise, target is the point itself.
 """
 target = point_or_mobject.get_critical_point(aligned_edge)
 else:
 target = point_or_mobject
 point_to_align = self.get_critical_point(aligned_edge)
 self.shift((target - point_to_align) * coor_mask)
 return self

def replace(self, mobject, dim_to_match=0, stretch=False):
 if not mobject.get_num_points() and not mobject.submobjects:
 raise Warning("Attempting to replace mobject with no points")
 if stretch:
 self.stretch_to_fit_width(mobject.width)
 self.stretch_to_fit_height(mobject.height)
 else:
 self.rescale_to_fit(
 mobject.length_over_dim(dim_to_match),
 dim_to_match,
 stretch=False,
)
 self.shift(mobject.get_center() - self.get_center())
 return self

def surround(
 self,
 mobject: Mobject,
 dim_to_match=0,
 stretch=False,
 buff=MED_SMALL_BUFF,
):
 self.replace(mobject, dim_to_match, stretch)
 length = mobject.length_over_dim(dim_to_match)
 self.scale((length + buff) / length)
 return self

def put_start_and_end_on(self, start, end):
 curr_start, curr_end = self.get_start_and_end()
 curr_vect = curr_end - curr_start
 if np.all(curr_vect == 0):
 raise Exception("Cannot position endpoints of closed loop")
 target_vect = np.array(end) - np.array(start)
 axis = (
 normalize(np.cross(curr_vect, target_vect))
 if np.linalg.norm(np.cross(curr_vect, target_vect)) != 0
 else OUT
)
 self.scale(
 np.linalg.norm(target_vect) / np.linalg.norm(curr_vect),

```

```

 about_point=curr_start,
)
 self.rotate(
 angle_between_vectors(curr_vect, target_vect),
 about_point=curr_start,
 axis=axis,
)
 self.shift(start - curr_start)
 return self

Background rectangle
def add_background_rectangle(
 self, color: Colors | None = None, opacity: float = 0.75, **kwargs
):
 """Add a BackgroundRectangle as submobject.

 The BackgroundRectangle is added behind other submobjects.

 This can be used to increase the mobjects visibility in front of a noisy background.

 Parameters

 color
 The color of the BackgroundRectangle
 opacity
 The opacity of the BackgroundRectangle
 kwargs
 Additional keyword arguments passed to the BackgroundRectangle constructor

 Returns

 :class:`Mobject`
 ``self``

 See Also

 :meth:`add_to_back`
 :class:`~.BackgroundRectangle`

 """
 # TODO, this does not behave well when the mobject has points,
 # since it gets displayed on top
 from manim.mobject.geometry.shape_matchers import BackgroundRectangle

 self.background_rectangle = BackgroundRectangle(
 self, color=color, fill_opacity=opacity, **kwargs
)
 self.add_to_back(self.background_rectangle)
 return self

def add_background_rectangle_to_submobjects(self, **kwargs):
 for submobject in self.submobjects:
 submobject.add_background_rectangle(**kwargs)
 return self

def add_background_rectangle_to_family_members_with_points(self, **kwargs):
 for mob in self.family_members_with_points():
 mob.add_background_rectangle(**kwargs)
 return self

Color functions

def set_color(self, color: Color = YELLOW_C, family: bool = True):
 """Condition is function which takes in one arguments, (x, y, z).
 Here it just recurses to submobjects, but in subclasses this
 should be further implemented based on the the inner workings
 of color
 """
 if family:
 for submob in self.submobjects:
 submob.set_color(color, family=family)
 self.color = Color(color)
 return self

def set_color_by_gradient(self, *colors):
 self.set_submobject_colors_by_gradient(*colors)
 return self

def set_colors_by_radial_gradient(
 self,
 center=None,
 radius=1,
 inner_color=WHITE,
 outer_color=BLACK,
):
 self.set_submobject_colors_by_radial_gradient(
 center,
 radius,
 inner_color,
 outer_color,
)
 return self

def set_submobject_colors_by_gradient(self, *colors):
 if len(colors) == 0:
 raise ValueError("Need at least one color")
 elif len(colors) == 1:

```

```

 return self.set_color(*colors)

mobs = self.family_members_with_points()
new_colors = color_gradient(colors, len(mobs))

for mob, color in zip(mobs, new_colors):
 mob.set_color(color, family=False)
return self

def set_submobobject_colors_by_radial_gradient(
 self,
 center=None,
 radius=1,
 inner_color=WHITE,
 outer_color=BLACK,
):
 if center is None:
 center = self.get_center()

 for mob in self.family_members_with_points():
 t = np.linalg.norm(mob.get_center() - center) / radius
 t = min(t, 1)
 mob_color = interpolate_color(inner_color, outer_color, t)
 mob.set_color(mob_color, family=False)

 return self

def to_original_color(self):
 self.set_color(self.color)
 return self

def fade_to(self, color, alpha, family=True):
 if self.get_num_points() > 0:
 new_color = interpolate_color(self.get_color(), color, alpha)
 self.set_color(new_color, family=False)
 if family:
 for submob in self.submobjects:
 submob.fade_to(color, alpha)
 return self

def fade(self, darkness=0.5, family=True):
 if family:
 for submob in self.submobjects:
 submob.fade(darkness, family)
 return self

def get_color(self):
 """Returns the color of the :class:`~.Mobject`"""
 return self.color

##

def save_state(self):
 """Save the current state (position, color & size). Can be restored with :meth:`~.Mobject.restore`."""
 if hasattr(self, "saved_state"):
 # Prevent exponential growth of data
 self.saved_state = None
 self.saved_state = self.copy()

 return self

def restore(self):
 """Restores the state that was previously saved with :meth:`~.Mobject.save_state`."""
 if not hasattr(self, "saved_state") or self.saved_state is None:
 raise Exception("Trying to restore without having saved")
 self.become(self.saved_state)
 return self

def reduce_across_dimension(self, reduce_func, dim: int) -> float:
 """Find the min or max value from a dimension across all points in this and submobjects."""
 assert dim >= 0 and dim <= 2
 if len(self.submobjects) == 0 and len(self.points) == 0:
 # If we have no points and no submobjects, return 0 (e.g. center)
 return 0

 # If we do not have points (but do have submobjects)
 # use only the points from those.
 if len(self.points) == 0:
 rv = None
 else:
 # Otherwise, be sure to include our own points
 rv = reduce_func(self.points[:, dim])
 # Recursively ask submobjects (if any) for the biggest/
 # smallest dimension they have and compare it to the return value.
 for mobj in self.submobjects:
 value = mobj.reduce_across_dimension(reduce_func, dim)
 if rv is None:
 rv = value
 else:
 rv = reduce_func([value, rv])
 return rv

def nonempty_submobjects(self):
 return [
 submob
 for submob in self.submobjects
 if len(submob.submobjects) != 0 or len(submob.points) != 0
]

```

```

def get_merged_array(self, array_attr) -> np.ndarray:
 """Return all of a given attribute from this mobject and all submobjects.

 May contain duplicates; the order is in a depth-first (pre-order)
 traversal of the submobjects.
 """
 result = getattr(self, array_attr)
 for submob in self.submobjects:
 result = np.append(result, submob.get_merged_array(array_attr), axis=0)
 return result

def get_all_points(self) -> np.ndarray:
 """Return all points from this mobject and all submobjects.

 May contain duplicates; the order is in a depth-first (pre-order)
 traversal of the submobjects.
 """
 return self.get_merged_array("points")

Getters

def get_points_defining_boundary(self):
 return self.get_all_points()

def get_num_points(self):
 return len(self.points)

def get_extremum_along_dim(self, points=None, dim=0, key=0):
 if points is None:
 points = self.get_points_defining_boundary()
 values = points[:, dim]
 if key < 0:
 return np.min(values)
 elif key == 0:
 return (np.min(values) + np.max(values)) / 2
 else:
 return np.max(values)

def get_critical_point(self, direction):
 """Picture a box bounding the :class:`~.Mobject`. Such a box has
 9 'critical points': 4 corners, 4 edge center, the
 center. This returns one of them, along the given direction.

 ::

 sample = Arc(start_angle=PI/7, angle = PI/5)

 # These are all equivalent
 max_y_1 = sample.get_top()[1]
 max_y_2 = sample.get_critical_point(UP)[1]
 max_y_3 = sample.get_extremum_along_dim(dim=1, key=1)

 """
 result = np.zeros(self.dim)
 all_points = self.get_points_defining_boundary()
 if len(all_points) == 0:
 return result
 for dim in range(self.dim):
 result[dim] = self.get_extremum_along_dim(
 all_points,
 dim=dim,
 key=direction[dim],
)
 return result

Pseudonyms for more general get_critical_point method

def get_edge_center(self, direction) -> np.ndarray:
 """Get edge coordinates for certain direction."""
 return self.get_critical_point(direction)

def get_corner(self, direction) -> np.ndarray:
 """Get corner coordinates for certain direction."""
 return self.get_critical_point(direction)

def get_center(self) -> np.ndarray:
 """Get center coordinates"""
 return self.get_critical_point(np.zeros(self.dim))

def get_center_of_mass(self):
 return np.apply_along_axis(np.mean, 0, self.get_all_points())

def get_boundary_point(self, direction):
 all_points = self.get_points_defining_boundary()
 index = np.argmax(np.dot(all_points, np.array(direction).T))
 return all_points[index]

def get_midpoint(self) -> np.ndarray:
 """Get coordinates of the middle of the path that forms the :class:`~.Mobject`."""

Examples

.. manim:: AngleMidPoint
 :save_last_frame:

 class AngleMidPoint(Scene):
 def construct(self):
 line1 = Line(ORIGIN, 2*RIGHT)

```

```

 line2 = Line(ORIGIN, 2*RIGHT).rotate_about_origin(80*DEGREES)

 a = Angle(line1, line2, radius=1.5, other_angle=False)
 d = Dot(a.get_midpoint()).set_color(RED)

 self.add(line1, line2, a, d)
 self.wait()

 """
 return self.point_from_proportion(0.5)

def get_top(self) -> np.ndarray:
 """Get top coordinates of a box bounding the :class:`~.Mobject`"""
 return self.get_edge_center(UP)

def get_bottom(self) -> np.ndarray:
 """Get bottom coordinates of a box bounding the :class:`~.Mobject`"""
 return self.get_edge_center(DOWN)

def get_right(self) -> np.ndarray:
 """Get right coordinates of a box bounding the :class:`~.Mobject`"""
 return self.get_edge_center(RIGHT)

def get_left(self) -> np.ndarray:
 """Get left coordinates of a box bounding the :class:`~.Mobject`"""
 return self.get_edge_center(LEFT)

def get_zzenith(self) -> np.ndarray:
 """Get zenith coordinates of a box bounding a 3D :class:`~.Mobject`"""
 return self.get_edge_center(OUT)

def get_znadir(self) -> np.ndarray:
 """Get nadir (opposite the zenith) coordinates of a box bounding a 3D :class:`~.Mobject`"""
 return self.get_edge_center(IN)

def length_over_dim(self, dim):
 """Measure the length of an :class:`~.Mobject` in a certain direction."""
 return self.reduce_across_dimension(
 max,
 dim,
) - self.reduce_across_dimension(min, dim)

def get_coord(self, dim, direction=ORIGIN):
 """Meant to generalize ``get_x``, ``get_y`` and ``get_z``"""
 return self.get_extremum_along_dim(dim=dim, key=direction[dim])

def get_x(self, direction=ORIGIN) -> np.float64:
 """Returns x coordinate of the center of the :class:`~.Mobject` as ``float``"""
 return self.get_coord(0, direction)

def get_y(self, direction=ORIGIN) -> np.float64:
 """Returns y coordinate of the center of the :class:`~.Mobject` as ``float``"""
 return self.get_coord(1, direction)

def get_z(self, direction=ORIGIN) -> np.float64:
 """Returns z coordinate of the center of the :class:`~.Mobject` as ``float``"""
 return self.get_coord(2, direction)

def get_start(self):
 """Returns the point, where the stroke that surrounds the :class:`~.Mobject` starts."""
 self.throw_error_if_no_points()
 return np.array(self.points[0])

def get_end(self):
 """Returns the point, where the stroke that surrounds the :class:`~.Mobject` ends."""
 self.throw_error_if_no_points()
 return np.array(self.points[-1])

def get_start_and_end(self):
 """Returns starting and ending point of a stroke as a ``tuple``."""
 return self.get_start(), self.get_end()

def point_from_proportion(self, alpha):
 raise NotImplementedError("Please override in a child class.")

def proportion_from_point(self, point):
 raise NotImplementedError("Please override in a child class.")

def get_pieces(self, n_pieces):
 template = self.copy()
 template.submobjects = []
 alphas = np.linspace(0, 1, n_pieces + 1)
 return Group(
 *(
 template.copy().pointwise_become_partial(self, a1, a2)
 for a1, a2 in zip(alphas[:-1], alphas[1:])
)
)

def get_z_index_reference_point(self):
 # TODO, better place to define default z_index_group?
 z_index_group = getattr(self, "z_index_group", self)
 return z_index_group.get_center()

def has_points(self) -> bool:
 """Check if :class:`~.Mobject` contains points."""
 return len(self.points) > 0

def has_no_points(self) -> bool:

```

```

"""Check if :class:`~.Mobject` *does not* contains points."""
return not self.has_points()

Match other mobject properties

def match_color(self, mobject: Mobject):
 """Match the color with the color of another :class:`~.Mobject`."""
 return self.set_color(mobject.get_color())

def match_dim_size(self, mobject: Mobject, dim, **kwargs):
 """Match the specified dimension with the dimension of another :class:`~.Mobject`."""
 return self.rescale_to_fit(mobject.length_over_dim(dim), dim, **kwargs)

def match_width(self, mobject: Mobject, **kwargs):
 """Match the width with the width of another :class:`~.Mobject`."""
 return self.match_dim_size(mobject, 0, **kwargs)

def match_height(self, mobject: Mobject, **kwargs):
 """Match the height with the height of another :class:`~.Mobject`."""
 return self.match_dim_size(mobject, 1, **kwargs)

def match_depth(self, mobject: Mobject, **kwargs):
 """Match the depth with the depth of another :class:`~.Mobject`."""
 return self.match_dim_size(mobject, 2, **kwargs)

def match_coord(self, mobject: Mobject, dim, direction=ORIGIN):
 """Match the coordinates with the coordinates of another :class:`~.Mobject`."""
 return self.set_coord(
 mobject.get_coord(dim, direction),
 dim=dim,
 direction=direction,
)

def match_x(self, mobject: Mobject, direction=ORIGIN):
 """Match x coord. to the x coord. of another :class:`~.Mobject`."""
 return self.match_coord(mobject, 0, direction)

def match_y(self, mobject: Mobject, direction=ORIGIN):
 """Match y coord. to the x coord. of another :class:`~.Mobject`."""
 return self.match_coord(mobject, 1, direction)

def match_z(self, mobject: Mobject, direction=ORIGIN):
 """Match z coord. to the x coord. of another :class:`~.Mobject`."""
 return self.match_coord(mobject, 2, direction)

def align_to(
 self,
 mobject_or_point: Mobject | np.ndarray | list,
 direction=ORIGIN,
):
 """Aligns mobject to another :class:`~.Mobject` in a certain direction.

 Examples:
 mob1.align_to(mob2, UP) moves mob1 vertically so that its
 top edge lines up with mob2's top edge.
 """
 if isinstance(mobject_or_point, Mobject):
 point = mobject_or_point.get_critical_point(direction)
 else:
 point = mobject_or_point

 for dim in range(self.dim):
 if direction[dim] != 0:
 self.set_coord(point[dim], dim, direction)
 return self

Family matters

def __getitem__(self, value):
 self_list = self.split()
 if isinstance(value, slice):
 GroupClass = self.get_group_class()
 return GroupClass(*self_list.__getitem__(value))
 return self_list.__getitem__(value)

def __iter__(self):
 return iter(self.split())

def __len__(self):
 return len(self.split())

def get_group_class(self):
 return Group

@staticmethod
def get_mobject_type_class():
 """Return the base class of this mobject type."""
 return Mobject

def split(self):
 result = [self] if len(self.points) > 0 else []
 return result + self.submobjects

def get_family(self, recurse=True):
 sub_families = list(map(Mobject.get_family, self.submobjects))
 all_mobjects = [self] + list(it.chain(*sub_families))
 return remove_list_redundancies(all_mobjects)

def family_members_with_points(self):

```

```

 return [m for m in self.get_family() if m.get_num_points() > 0]

def arrange(
 self,
 direction: Sequence[float] = RIGHT,
 buff=DEFAULT_MOBJECT_TO_MOBJECT_BUFFER,
 center=True,
 **kwargs,
):
 """Sorts :class:`~.Mobject` next to each other on screen.

Examples

.. manim:: Example
 :save_last_frame:

 class Example(Scene):
 def construct(self):
 s1 = Square()
 s2 = Square()
 s3 = Square()
 s4 = Square()
 x = VGroup(s1, s2, s3, s4).set_x(0).arrange(buff=1.0)
 self.add(x)

 """
 for m1, m2 in zip(self.submobjects, self.submobjects[1:]):
 m2.next_to(m1, direction, buff, **kwargs)
 if center:
 self.center()
 return self

def arrange_in_grid(
 self,
 rows: int | None = None,
 cols: int | None = None,
 buff: float | tuple[float, float] = MED_SMALL_BUFF,
 cell_alignment: np.ndarray = ORIGIN,
 row_alignments: str | None = None, # "ucd"
 col_alignments: str | None = None, # "lcr"
 row_heights: Iterable[float | None] | None = None,
 col_widths: Iterable[float | None] | None = None,
 flow_order: str = "rd",
 **kwargs,
):
 """Arrange submobjects in a grid.

Parameters

rows
 The number of rows in the grid.
cols
 The number of columns in the grid.
buff
 The gap between grid cells. To specify a different buffer in the horizontal and vertical directions, a tuple of two values can be given - ``(row, col)``.
cell_alignment
 The way each submobject is aligned in its grid cell.
row_alignments
 The vertical alignment for each row (top to bottom). Accepts the following characters: ``"u"`` - up, ``"c"`` - center, ``"d"`` - down.
col_alignments
 The horizontal alignment for each column (left to right). Accepts the following characters ``"l"`` - left, ``"c"`` - center, ``"r"`` - right.
row_heights
 Defines a list of heights for certain rows (top to bottom). If the list contains ``None``, the corresponding row will fit its height automatically based on the highest element in that row.
col_widths
 Defines a list of widths for certain columns (left to right). If the list contains ``None``, the corresponding column will fit its width automatically based on the widest element in that column.
flow_order
 The order in which submobjects fill the grid. Can be one of the following values: "rd", "dr", "ld", "dl", "ru", "ur", "lu", "ul". ("rd" -> fill rightwards then downwards)

Returns

:class:`Mobject`
 ``self``

Raises

ValueError
 If ``rows`` and ``cols`` are too small to fit all submobjects.
ValueError
 If :code:`cols`, :code:`col_alignments` and :code:`col_widths` or :code:`rows`, :code:`row_alignments` and :code:`row_heights` have mismatching sizes.

Notes

If only one of ``cols`` and ``rows`` is set implicitly, the other one will be chosen big enough to fit all submobjects. If neither is set, they will be chosen to be about the same, tending towards ``cols`` > ``rows`` (simply because videos are wider than they are high).

If both ``cell_alignment`` and ``row_alignments`` / ``col_alignments`` are defined, the latter has higher priority.

Examples

```

```

.. manim:: ExampleBoxes
 :save_last_frame:

 class ExampleBoxes(Scene):
 def construct(self):
 boxes=VGroup(*[Square() for s in range(0,6)])
 boxes.arrange_in_grid(rows=2, buff=0.1)
 self.add(boxes)

.. manim:: ArrangeInGrid
 :save_last_frame:

 class ArrangeInGrid(Scene):
 def construct(self):
 boxes = VGroup(*[
 Rectangle(WHITE, 0.5, 0.5).add(Text(str(i+1)).scale(0.5))
 for i in range(24)
])
 self.add(boxes)

 boxes.arrange_in_grid(
 buff=(0.25,0.5),
 col_alignments="lccccr",
 row_alignments="uccd",
 col_widths=[1, *[None]*4, 1],
 row_heights=[1, None, None, 1],
 flow_order="dr"
)

"""

from manim.mobject.geometry.line import Line

mobs = self.submobjects.copy()
start_pos = self.get_center()

get cols / rows values if given (implicitly)
def init_size(num, alignments, sizes):
 if num is not None:
 return num
 if alignments is not None:
 return len(alignments)
 if sizes is not None:
 return len(sizes)

cols = init_size(cols, col_alignments, col_widths)
rows = init_size(rows, row_alignments, row_heights)

calculate rows cols
if rows is None and cols is None:
 cols = ceil(np.sqrt(len(mobs)))
 # make the grid as close to quadratic as possible.
 # choosing cols first can results in cols>rows.
 # This is favored over rows>cols since in general
 # the scene is wider than high.
if rows is None:
 rows = ceil(len(mobs) / cols)
if cols is None:
 cols = ceil(len(mobs) / rows)
if rows * cols < len(mobs):
 raise ValueError("Too few rows and columns to fit all submobjects.")
rows and cols are now finally valid.

if isinstance(buff, tuple):
 buff_x = buff[0]
 buff_y = buff[1]
else:
 buff_x = buff_y = buff

Initialize alignments correctly
def init_alignments(alignments, num, mapping, name, dir):
 if alignments is None:
 # Use cell_alignment as fallback
 return [cell_alignment * dir] * num
 if len(alignments) != num:
 raise ValueError(f"{name}_alignments has a mismatching size.")
 alignments = list(alignments)
 for i in range(num):
 alignments[i] = mapping[alignments[i]]
 return alignments

row_alignments = init_alignments(
 row_alignments,
 rows,
 {"u": UP, "c": ORIGIN, "d": DOWN},
 "row",
 RIGHT,
)
col_alignments = init_alignments(
 col_alignments,
 cols,
 {"l": LEFT, "c": ORIGIN, "r": RIGHT},
 "col",
 UP,
)
Now row_alignment[r] + col_alignment[c] is the alignment in cell [r][c]

mapper = {

```

```

 "dr": lambda r, c: (rows - r - 1) + c * rows,
 "dl": lambda r, c: (rows - r - 1) + (cols - c - 1) * rows,
 "ur": lambda r, c: r + c * rows,
 "ul": lambda r, c: r + (cols - c - 1) * rows,
 "rd": lambda r, c: (rows - r - 1) * cols + c,
 "ld": lambda r, c: (rows - r - 1) * cols + (cols - c - 1),
 "ru": lambda r, c: r * cols + c,
 "lu": lambda r, c: r * cols + (cols - c - 1),
}
if flow_order not in mapper:
 raise ValueError(
 'flow_order must be one of the following values: "dr", "rd", "ld" "dl", "ru", "ur", "lu", "ul".',
)
flow_order = mapper[flow_order]

Reverse row_alignments and row_heights. Necessary since the
grid filling is handled bottom up for simplicity reasons.
def reverse(maybe_list):
 if maybe_list is not None:
 maybe_list = list(maybe_list)
 maybe_list.reverse()
 return maybe_list

row_alignments = reverse(row_alignments)
row_heights = reverse(row_heights)

placeholder = Mobject()
Used to fill up the grid temporarily, doesn't get added to the scene.
In this case a Mobject is better than None since it has width and height
properties of 0.

mobs.extend([placeholder] * (rows * cols - len(mobs)))
grid = [[mobs[flow_order(r, c)] for c in range(cols)] for r in range(rows)]

measured_heights = [
 max(grid[r][c].height for c in range(cols)) for r in range(rows)
]
measured_widths = [
 max(grid[r][c].width for r in range(rows)) for c in range(cols)
]

Initialize row_heights / col_widths correctly using measurements as fallback
def init_sizes(sizes, num, measures, name):
 if sizes is None:
 sizes = [None] * num
 if len(sizes) != num:
 raise ValueError(f"{name} has a mismatching size.")
 return [
 sizes[i] if sizes[i] is not None else measures[i] for i in range(num)
]

heights = init_sizes(row_heights, rows, measured_heights, "row_heights")
widths = init_sizes(col_widths, cols, measured_widths, "col_widths")

x, y = 0, 0
for r in range(rows):
 x = 0
 for c in range(cols):
 if grid[r][c] is not placeholder:
 alignment = row_alignments[r] + col_alignments[c]
 line = Line(
 x * RIGHT + y * UP,
 (x + widths[c]) * RIGHT + (y + heights[r]) * UP,
)
 # Use a mobject to avoid rewriting align inside
 # box code that Mobject.move_to(Mobject) already
 # includes.

 grid[r][c].move_to(line, alignment)
 x += widths[c] + buff_x
 y += heights[r] + buff_y

 self.move_to(start_pos)
 return self

def sort(self, point_to_num_func=lambda p: p[0], submob_func=None):
 """Sorts the list of :attr:`submobjects` by a function defined by ``submob_func``."""
 if submob_func is None:

 def submob_func(m):
 return point_to_num_func(m.get_center())

 self.submobjects.sort(key=submob_func)
 return self

def shuffle(self, recursive=False):
 """Shuffles the list of :attr:`submobjects`."""
 if recursive:
 for submob in self.submobjects:
 submob.shuffle(recursive=True)
 random.shuffle(self.submobjects)

def invert(self, recursive=False):
 """Inverts the list of :attr:`submobjects`.

Parameters

recursive
 If ``True``, all submobject lists of this mobject's family are inverted.

```

```

Examples

.. manim:: InvertSumobjectsExample

 class InvertSumobjectsExample(Scene):
 def construct(self):
 s = VGroup(*[Dot().shift(i*0.1*RIGHT) for i in range(-20,20)])
 s2 = s.copy()
 s2.invert()
 s2.shift(DOWN)
 self.play(Write(s), Write(s2))
 """
 if recursive:
 for submob in self.submobjects:
 submob.invert(recursive=True)
 list.reverse(self.submobjects)

Just here to keep from breaking old scenes.
def arrange_submobjects(self, *args, **kwargs):
 """Arrange the position of :attr:`submobjects` with a small buffer.

Examples

.. manim:: ArrangeSumobjectsExample
:save_last_frame:

 class ArrangeSumobjectsExample(Scene):
 def construct(self):
 s= VGroup(*[Dot().shift(i*0.1*RIGHT*np.random.uniform(-1,1)+UP*np.random.uniform(-1,1)) for i in range(0,15)])
 s.shift(UP).set_color(BLUE)
 s2= s.copy().set_color(RED)
 s2.arrange_submobjects()
 s2.shift(DOWN)
 self.add(s,s2)

 """
 return self.arrange(*args, **kwargs)

def sort_submobjects(self, *args, **kwargs):
 """Sort the :attr:`submobjects`"""
 return self.sort(*args, **kwargs)

def shuffle_submobjects(self, *args, **kwargs):
 """Shuffles the order of :attr:`submobjects`

Examples

.. manim:: ShuffleSubmobjectsExample

 class ShuffleSubmobjectsExample(Scene):
 def construct(self):
 s= VGroup(*[Dot().shift(i*0.1*RIGHT) for i in range(-20,20)])
 s2= s.copy()
 s2.shuffle_submobjects()
 s2.shift(DOWN)
 self.play(Write(s), Write(s2))
 """
 return self.shuffle(*args, **kwargs)

Alignment
def align_data(self, mobject: Mobject, skip_point_alignment: bool = False):
 """Aligns the data of this mobject with another mobject.

Afterwards, the two mobjects will have the same number of submobjects
(see :meth:`.align_submobjects`), the same parent structure (see
:meth:`.null_point_align`). If ``skip_point_alignment`` is false,
they will also have the same number of points (see :meth:`.align_points`).

Parameters

mobject
 The other mobject this mobject should be aligned to.
skip_point_alignment
 Controls whether or not the computationally expensive
 point alignment is skipped (default: False).
"""
self.null_point_align(mobject)
self.align_submobjects(mobject)
if not skip_point_alignment:
 self.align_points(mobject)
Recurse
for m1, m2 in zip(self.submobjects, mobject.submobjects):
 m1.align_data(m2)

def get_point_mobject(self, center=None):
 """The simplest :class:`~.Mobject` to be transformed to or from self.
Should by a point of the appropriate type
"""
msg = f"get_point_mobject not implemented for {self.__class__.__name__}"
raise NotImplementedError(msg)

def align_points(self, mobject):
 count1 = self.get_num_points()
 count2 = mobject.get_num_points()
 if count1 < count2:

```

```

 self.align_points_with_larger(mobject)
elif count2 < count1:
 mobject.align_points_with_larger(self)
return self

def align_points_with_larger(self, larger_mobject):
 raise NotImplementedError("Please override in a child class.")

def align_submobjects(self, mobject):
 mob1 = self
 mob2 = mobject
 n1 = len(mob1.submobjects)
 n2 = len(mob2.submobjects)
 mob1.add_n_more_submobjects(max(0, n2 - n1))
 mob2.add_n_more_submobjects(max(0, n1 - n2))
 return self

def null_point_align(self, mobject: Mobject):
 """If a :class:`~.Mobject` with points is being aligned to one without, treat both as groups, and push the one with points into its own submobjects list.

 Returns

 :class:`Mobject` ``self``
 """
 for m1, m2 in (self, mobject), (mobject, self):
 if m1.has_no_points() and m2.has_points():
 m2.push_self_into_submobjects()
 return self

def push_self_into_submobjects(self):
 copy = self.copy()
 copy.submobjects = []
 self.reset_points()
 self.add(copy)
 return self

def add_n_more_submobjects(self, n):
 if n == 0:
 return

 curr = len(self.submobjects)
 if curr == 0:
 # If empty, simply add n point mobjects
 self.submobjects = [self.get_point_mobject() for k in range(n)]
 return

 target = curr + n
 # TODO, factor this out to utils so as to reuse
 # with VMobject.insert_n_curves
 repeat_indices = (np.arange(target) * curr) // target
 split_factors = [sum(repeat_indices == i) for i in range(curr)]
 new_submobs = []
 for submob, sf in zip(self.submobjects, split_factors):
 new_submobs.append(submob)
 for _ in range(1, sf):
 new_submobs.append(submob.copy().fade(1))
 self.submobjects = new_submobs
 return self

def repeat_submobject(self, submob):
 return submob.copy()

def interpolate(self, mobject1, mobject2, alpha, path_func=straight_path()):
 """Turns this :class:`~.Mobject` into an interpolation between ``mobject1`` and ``mobject2``.

 Examples

 .. manim:: DotInterpolation
 :save_last_frame:

 class DotInterpolation(Scene):
 def construct(self):
 dotR = Dot(color=DARK_GREY)
 dotR.shift(2 * RIGHT)
 dotL = Dot(color=WHITE)
 dotL.shift(2 * LEFT)

 dotMiddle = VMobject().interpolate(dotL, dotR, alpha=0.3)

 self.add(dotL, dotR, dotMiddle)

 self.points = path_func(mobject1.points, mobject2.points, alpha)
 self.interpolate_color(mobject1, mobject2, alpha)
 return self

 def interpolate_color(self, mobject1, mobject2, alpha):
 raise NotImplementedError("Please override in a child class.")

 def become(
 self,
 mobject: Mobject,
 copy_submobjects: bool = True,
 match_height: bool = False,

```

```

match_width: bool = False,
match_depth: bool = False,
match_center: bool = False,
stretch: bool = False,
):
 """Edit points, colors and subobjects to be identical
 to another :class:`~.Mobject`"""

 .. note::

 If both match_height and match_width are ``True`` then the transformed :class:`~.Mobject`
 will match the height first and then the width

 Parameters

 match_height
 If ``True``, then the transformed :class:`~.Mobject` will match the height of the original
 match_width
 If ``True``, then the transformed :class:`~.Mobject` will match the width of the original
 match_depth
 If ``True``, then the transformed :class:`~.Mobject` will match the depth of the original
 match_center
 If ``True``, then the transformed :class:`~.Mobject` will match the center of the original
 stretch
 If ``True``, then the transformed :class:`~.Mobject` will stretch to fit the proportions of the original

 Examples

 .. manim:: BecomeScene

 class BecomeScene(Scene):
 def construct(self):
 circ = Circle(fill_color=RED, fill_opacity=0.8)
 square = Square(fill_color=BLUE, fill_opacity=0.2)
 self.add(circ)
 self.wait(0.5)
 circ.become(square)
 self.wait(0.5)

 """
 if stretch:
 mobject.stretch_to_fit_height(self.height)
 mobject.stretch_to_fit_width(self.width)
 mobject.stretch_to_fit_depth(self.depth)
 else:
 if match_height:
 mobject.match_height(self)
 if match_width:
 mobject.match_width(self)
 if match_depth:
 mobject.match_depth(self)

 if match_center:
 mobject.move_to(self.get_center())

 self.align_data(mobject, skip_point_alignment=True)
 for sm1, sm2 in zip(self.get_family(), mobject.get_family()):
 sm1.points = np.array(sm2.points)
 sm1.interpolate_color(sm1, sm2, 1)
 return self

 def match_points(self, mobject: Mobject, copy_subobjects: bool = True):
 """Edit points, positions, and subobjects to be identical
 to another :class:`~.Mobject`, while keeping the style unchanged.

 Examples

 .. manim:: MatchPointsScene

 class MatchPointsScene(Scene):
 def construct(self):
 circ = Circle(fill_color=RED, fill_opacity=0.8)
 square = Square(fill_color=BLUE, fill_opacity=0.2)
 self.add(circ)
 self.wait(0.5)
 self.play(circ.animate.match_points(square))
 self.wait(0.5)

 """
 for sm1, sm2 in zip(self.get_family(), mobject.get_family()):
 sm1.points = np.array(sm2.points)
 return self

 # Errors
 def throw_error_if_no_points(self):
 if self.has_no_points():
 caller_name = sys._getframe(1).f_code.co_name
 raise Exception(
 f"Cannot call Mobject.{caller_name} for a Mobject with no points",
)

 # About z-index
 def set_z_index(
 self,
 z_index_value: float,
 family: bool = True,
) -> VMobject:
 """Sets the :class:`~.Mobject`'s :attr:`z_index` to the value specified in `z_index_value`."""

 Parameters

```

```

z_index_value
 The new value of :attr:`z_index` set.
family
 If ``True``, the :attr:`z_index` value of all submobjects is also set.

>Returns

:class:`Mobject`
 The Mobject itself, after :attr:`z_index` is set. For chaining purposes. (Returns `self`.)

Examples

.. manim:: SetZIndex
 :save_last_frame:

 class SetZIndex(Scene):
 def construct(self):
 text = Text('z_index = 3', color = PURE_RED).shift(UP).set_z_index(3)
 square = Square(2, fill_opacity=1).set_z_index(2)
 tex = Tex(r'zIndex = 1', color = PURE_BLUE).shift(DOWN).set_z_index(1)
 circle = Circle(radius = 1.7, color = GREEN, fill_opacity = 1) # z_index = 0

 # Displaying order is now defined by z_index values
 self.add(text)
 self.add(square)
 self.add(tex)
 self.add(circle)

 """
 if family:
 for submob in self.submobjects:
 submob.set_z_index(z_index_value, family=family)
 self.z_index = z_index_value
 return self

def set_z_index_by_z_coordinate(self):
 """Sets the :class:`~.Mobject`'s z coordinate to the value of :attr:`z_index`.

>Returns

:class:`Mobject`
 The Mobject itself, after :attr:`z_index` is set. (Returns `self`.)

 z_coord = self.get_center()[-1]
 self.set_z_index(z_coord)
 return self

class Group(Mobject, metaclass=ConvertToOpenGL):
 """Groups together multiple :class:`Mobjects <.Mobject>`."""

Notes

When adding the same mobject more than once, repetitions are ignored.
Use :meth:`.Mobject.copy` to create a separate copy which can then
be added to the group.
"""

def __init__(self, *mobjects, **kwargs):
 super().__init__(**kwargs)
 self.add(*mobjects)

class _AnimationBuilder:
 def __init__(self, mobject):
 self.mobject = mobject
 self.mobject.generate_target()

 self.overridden_animation = None
 self.is_chaining = False
 self.methods = []

 # Whether animation args can be passed
 self.cannot_pass_args = False
 self.anim_args = {}

 def __call__(self, **kwargs):
 if self.cannot_pass_args:
 raise ValueError(
 "Animation arguments must be passed before accessing methods and can only be passed once",
)

 self.anim_args = kwargs
 self.cannot_pass_args = True

 return self

def __getattr__(self, method_name):
 method = getattr(self.mobject.target, method_name)
 has_overridden_animation = hasattr(method, "_override_animate")

 if (self.is_chaining and has_overridden_animation) or self.overridden_animation:
 raise NotImplementedError(
 "Method chaining is currently not supported for "
 "overridden animations",
)

 def update_target(*method_args, **method_kwargs):
 if has_overridden_animation:

```

```

 self.overridden_animation = method._override_animate(
 self.mobject,
 *method_args,
 anim_args=self.anim_args,
 **method_kwargs,
)
 else:
 self.methods.append([method, method_args, method_kwargs])
 method(*method_args, **method_kwargs)
 return self

 self.is_chaining = True
 self.cannot_pass_args = True

 return update_target

def build(self):
 from ..animation.transform import _MethodAnimation

 if self.overridden_animation:
 anim = self.overridden_animation
 else:
 anim = _MethodAnimation(self.mobject, self.methods)

 for attr, value in self.anim_args.items():
 setattr(anim, attr, value)

 return anim

def override_animate(method):
 """Decorator for overriding method animations.

 This allows to specify a method (returning an :class:`~.Animation`)
 which is called when the decorated method is used with the ```.animate```
 syntax for animating the application of a method.

 .. seealso::

 :attr:`Mobject.animate`

 .. note::

 Overridden methods cannot be combined with normal or other overridden
 methods using method chaining with the ```.animate```` syntax.

 Examples

 .. manim:: AnimationOverrideExample

 class CircleWithContent(VGroup):
 def __init__(self, content):
 super().__init__()
 self.circle = Circle()
 self.content = content
 self.add(self.circle, content)
 content.move_to(self.circle.get_center())

 def clear_content(self):
 self.remove(self.content)
 self.content = None

 @override_animate(clear_content)
 def _clear_content_animation(self, anim_args=None):
 if anim_args is None:
 anim_args = {}
 anim = Uncreate(self.content, **anim_args)
 self.clear_content()
 return anim

 class AnimationOverrideExample(Scene):
 def construct(self):
 t = Text("hello!")
 my_mobject = CircleWithContent(t)
 self.play(Create(my_mobject))
 self.play(my_mobject.animate.clear_content())
 self.wait()

 """

 def decorator(animation_method):
 method._override_animate = animation_method
 return animation_method

 return decorator

```

# opengl\_geometry.py

```
from __future__ import annotations

import numpy as np

from manim.constants import *
from manim.mobject.mobject import Mobject
from manim.mobject.opengl_vectorized_mobject import (
 OpenGLDashedVMobject,
 OpenGLVGroup,
 OpenGLVMobject,
)
from manim.utils.color import *
from manim.utils.iterables import adjacent_n_tuples, adjacent_pairs
from manim.utils.simple_functions import clip
from manim.utils.space_ops import (
 angle_between_vectors,
 angle_of_vector,
 compass_directions,
 find_intersection,
 normalize,
 rotate_vector,
 rotation_matrix_transpose,
)
DEFAULT_DOT_RADIUS = 0.08
DEFAULT_SMALL_DOT_RADIUS = 0.04
DEFAULT_DASH_LENGTH = 0.05
DEFAULT_ARROW_TIP_LENGTH = 0.35
DEFAULT_ARROW_TIP_WIDTH = 0.35

class OpenGLTipableVMobject(OpenGLVMobject):
 """
 Meant for shared functionality between Arc and Line.
 Functionality can be classified broadly into these groups:

 * Adding, Creating, Modifying tips
 - add_tip calls create_tip, before pushing the new tip
 into the TipableVMobject's list of submobjects
 - stylistic and positional configuration

 * Checking for tips
 - Boolean checks for whether the TipableVMobject has a tip
 and a starting tip

 * Getters
 - Straightforward accessors, returning information pertaining
 to the TipableVMobject instance's tip(s), its length etc
 """
 # Adding, Creating, Modifying tips

 def __init__(
 self,
 tip_length=DEFAULT_ARROW_TIP_LENGTH,
 normal_vector=OUT,
 tip_config={},
 **kwargs,
):
 self.tip_length = tip_length
 self.normal_vector = normal_vector
 self.tip_config = tip_config
 super().__init__(**kwargs)

 def add_tip(self, at_start=False, **kwargs):
 """
 Adds a tip to the TipableVMobject instance, recognising
 that the endpoints might need to be switched if it's
 a 'starting tip' or not.
 """
 tip = self.create_tip(at_start, **kwargs)
 self.reset_endpoints_based_on_tip(tip, at_start)
 self.assign_tip_attr(tip, at_start)
 self.add(tip)
 return self

 def create_tip(self, at_start=False, **kwargs):
 """
 Styliases the tip, positions it spacially, and returns
 the newly instantiated tip to the caller.
 """
 pass
```

```

tip = self.get_unpositioned_tip(**kwargs)
self.position_tip(tip, at_start)
return tip

def get_unpositioned_tip(self, **kwargs):
 """
 Returns a tip that has been stylistically configured,
 but has not yet been given a position in space.
 """
 config = {}
 config.update(self.tip_config)
 config.update(kwargs)
 return OpenGLArrowTip(**config)

def position_tip(self, tip, at_start=False):
 # Last two control points, defining both
 # the end, and the tangency direction
 if at_start:
 anchor = self.get_start()
 handle = self.get_first_handle()
 else:
 handle = self.get_last_handle()
 anchor = self.get_end()
 tip.rotate(angle_of_vector(handle - anchor) - PI - tip.get_angle())
 tip.shift(anchor - tip.get_tip_point())
 return tip

def reset_endpoints_based_on_tip(self, tip, at_start):
 if self.get_length() == 0:
 # Zero length, put_start_and_end_on wouldn't
 # work
 return self

 if at_start:
 start = tip.get_base()
 end = self.get_end()
 else:
 start = self.get_start()
 end = tip.get_base()
 self.put_start_and_end_on(start, end)
 return self

def assign_tip_attr(self, tip, at_start):
 if at_start:
 self.start_tip = tip
 else:
 self.tip = tip
 return self

Checking for tips
def has_tip(self):
 return hasattr(self, "tip") and self.tip in self

def has_start_tip(self):
 return hasattr(self, "start_tip") and self.start_tip in self

Getters
def pop_tips(self):
 start, end = self.get_start_and_end()
 result = OpenGLGroup()
 if self.has_tip():
 result.add(self.tip)
 self.remove(self.tip)
 if self.has_start_tip():
 result.add(self.start_tip)
 self.remove(self.start_tip)
 self.put_start_and_end_on(start, end)
 return result

def get_tips(self):
 """
 Returns a VGroup (collection of VMobjects) containing
 the TipableVMObject instance's tips.
 """
 result = OpenGLGroup()
 if hasattr(self, "tip"):
 result.add(self.tip)
 if hasattr(self, "start_tip"):
 result.add(self.start_tip)
 return result

def get_tip(self):
 """Returns the TipableVMobject instance's (first) tip,
 otherwise throws an exception."""
 tips = self.get_tips()
 if len(tips) == 0:
 raise Exception("tip not found")

```

```

 else:
 return tips[0]

 def get_default_tip_length(self):
 return self.tip_length

 def get_first_handle(self):
 return self.points[1]

 def get_last_handle(self):
 return self.points[-2]

 def get_end(self):
 if self.has_tip():
 return self.tip.get_start()
 else:
 return super().get_end()

 def get_start(self):
 if self.has_start_tip():
 return self.start_tip.get_start()
 else:
 return super().get_start()

 def get_length(self):
 start, end = self.get_start_and_end()
 return np.linalg.norm(start - end)

class OpenGLArc(OpenGLTipableVMobject):
 def __init__(self,
 start_angle=0,
 angle=TAU / 4,
 radius=1.0,
 n_components=8,
 arc_center=ORIGIN,
 **kwargs,
):
 self.start_angle = start_angle
 self.angle = angle
 self.radius = radius
 self.n_components = n_components
 self.arc_center = arc_center
 super().__init__(self, **kwargs)
 self.orientation = -1

 def init_points(self):
 self.set_points(
 OpenGLArc.create_quadratic_bezier_points(
 angle=self.angle,
 start_angle=self.start_angle,
 n_components=self.n_components,
),
)
 # To maintain proper orientation for fill shaders.
 self.scale(self.radius, about_point=ORIGIN)
 self.shift(self.arc_center)

 @staticmethod
 def create_quadratic_bezier_points(angle, start_angle=0, n_components=8):
 samples = np.array([
 [np.cos(a), np.sin(a), 0]
 for a in np.linspace(
 start_angle,
 start_angle + angle,
 2 * n_components + 1,
)
],
)
 theta = angle / n_components
 samples[1::2] /= np.cos(theta / 2)

 points = np.zeros((3 * n_components, 3))
 points[0::3] = samples[0:-1:2]
 points[1::3] = samples[1::2]
 points[2::3] = samples[2::2]
 return points

 def get_arc_center(self):
 """
 Looks at the normals to the first two anchors, and finds their intersection points
 """
 # First two anchors and handles
 a1, h, a2 = self.points[:3]

```

```

Tangent vectors
t1 = h - a1
t2 = h - a2
Normals
n1 = rotate_vector(t1, TAU / 4)
n2 = rotate_vector(t2, TAU / 4)
return find_intersection(a1, n1, a2, n2)

def get_start_angle(self):
 angle = angle_of_vector(self.get_start() - self.get_arc_center())
 return angle % TAU

def get_stop_angle(self):
 angle = angle_of_vector(self.get_end() - self.get_arc_center())
 return angle % TAU

def move_arc_center_to(self, point):
 self.shift(point - self.get_arc_center())
 return self

class OpenGLArcBetweenPoints(OpenGLArc):
 def __init__(self, start, end, angle=TAU / 4, **kwargs):
 super().__init__(angle=angle, **kwargs)
 if angle == 0:
 self.set_points_as_corners([LEFT, RIGHT])
 self.put_start_and_end_on(start, end)

class OpenGLCurvedArrow(OpenGLArcBetweenPoints):
 def __init__(self, start_point, end_point, **kwargs):
 super().__init__(start_point, end_point, **kwargs)
 self.add_tip()

class OpenGLCurvedDoubleArrow(OpenGLCurvedArrow):
 def __init__(self, start_point, end_point, **kwargs):
 super().__init__(start_point, end_point, **kwargs)
 self.add_tip(at_start=True)

class OpenGLCircle(OpenGLArc):
 def __init__(self, color=RED, **kwargs):
 super().__init__(0, TAU, color=color, **kwargs)

 def surround(self, mobject, dim_to_match=0, stretch=False, buff=MED_SMALL_BUFF):
 # Ignores dim_to_match and stretch; result will always be a circle
 # TODO: Perhaps create an ellipse class to handle single-dimension stretching

 self.replace(mobject, dim_to_match, stretch)
 self.stretch((self.get_width() + 2 * buff) / self.get_width(), 0)
 self.stretch((self.get_height() + 2 * buff) / self.get_height(), 1)

 def point_at_angle(self, angle):
 start_angle = self.get_start_angle()
 return self.point_from_proportion((angle - start_angle) / TAU)

class OpenGLDot(OpenGLCircle):
 def __init__(self,
 point=ORIGIN,
 radius=DEFAULT_DOT_RADIUS,
 stroke_width=0,
 fill_opacity=1.0,
 color=WHITE,
 **kwargs,
):
 super().__init__(
 arc_center=point,
 radius=radius,
 stroke_width=stroke_width,
 fill_opacity=fill_opacity,
 color=color,
 **kwargs,
)

class OpenGLEllipse(OpenGLCircle):
 def __init__(self, width=2, height=1, **kwargs):
 super().__init__(**kwargs)
 self.set_width(width, stretch=True)
 self.set_height(height, stretch=True)

class OpenGLAnnularSector(OpenGLArc):
 def __init__(self,

```

```

self,
inner_radius=1,
outer_radius=2,
angle=TAU / 4,
start_angle=0,
fill_opacity=1,
stroke_width=0,
color=WHITE,
**kwargs,
):
 self.inner_radius = inner_radius
 self.outer_radius = outer_radius
 super().__init__(
 start_angle=start_angle,
 angle=angle,
 fill_opacity=fill_opacity,
 stroke_width=stroke_width,
 color=color,
 **kwargs,
)

def init_points(self):
 inner_arc, outer_arc = (
 OpenGLArc(
 start_angle=self.start_angle,
 angle=self.angle,
 radius=radius,
 arc_center=self.arc_center,
)
 for radius in (self.inner_radius, self.outer_radius)
)
 outer_arc.reverse_points()
 self.append_points(inner_arc.points)
 self.add_line_to(outer_arc.points[0])
 self.append_points(outer_arc.points)
 self.add_line_to(inner_arc.points[0])

class OpenGLSector(OpenGLAnnularSector):
 def __init__(self, outer_radius=1, inner_radius=0, **kwargs):
 super().__init__(inner_radius=inner_radius, outer_radius=outer_radius, **kwargs)

class OpenGLAnnulus(OpenGLCircle):
 def __init__(_
 self,
 inner_radius=1,
 outer_radius=2,
 fill_opacity=1,
 stroke_width=0,
 color=WHITE,
 mark_paths_closed=False,
 **kwargs,
):
 self.mark_paths_closed = mark_paths_closed # is this even used?
 self.inner_radius = inner_radius
 self.outer_radius = outer_radius
 super().__init__(
 fill_opacity=fill_opacity, stroke_width=stroke_width, color=color, **kwargs
)

 def init_points(self):
 self.radius = self.outer_radius
 outer_circle = OpenGLCircle(radius=self.outer_radius)
 inner_circle = OpenGLCircle(radius=self.inner_radius)
 inner_circle.reverse_points()
 self.append_points(outer_circle.points)
 self.append_points(inner_circle.points)
 self.shift(self.arc_center)

class OpenGLLine(OpenGLTipableVMOBJECT):
 def __init__(self, start=LEFT, end=RIGHT, buff=0, path_arc=0, **kwargs):
 self.dim = 3
 self.buff = buff
 self.path_arc = path_arc
 self.set_start_and_end_attrs(start, end)
 super().__init__(**kwargs)

 def init_points(self):
 self.set_points_by_ends(self.start, self.end, self.buff, self.path_arc)

 def set_points_by_ends(self, start, end, buff=0, path_arc=0):
 if path_arc:
 self.set_points(OpenGLArc.create_quadratic_bezier_points(path_arc))
 self.put_start_and_end_on(start, end)
 else:

```

```

 self.set_points_as_corners([start, end])
 self.account_for_buff(self.buff)

def set_path_arc(self, new_value):
 self.path_arc = new_value
 self.init_points()

def account_for_buff(self, buff):
 if buff == 0:
 return
 #
 if self.path_arc == 0:
 length = self.get_length()
 else:
 length = self.get_arc_length()
 #
 if length < 2 * buff:
 return
 buff_prop = buff / length
 self.pointwise_become_partial(self, buff_prop, 1 - buff_prop)
 return self

def set_start_and_end_attrs(self, start, end):
 # If either start or end are Mobjects, this
 # gives their centers
 rough_start = self.pointify(start)
 rough_end = self.pointify(end)
 vect = normalize(rough_end - rough_start)
 # Now that we know the direction between them,
 # we can find the appropriate boundary point from
 # start and end, if they're mobjects
 self.start = self.pointify(start, vect) + self.buff * vect
 self.end = self.pointify(end, -vect) - self.buff * vect

def pointify(self, mob_or_point, direction=None):
 """
 Take an argument passed into Line (or subclass) and turn
 it into a 3d point.
 """
 if isinstance(mob_or_point, Mobject):
 mob = mob_or_point
 if direction is None:
 return mob.get_center()
 else:
 return mob.get_continuous_bounding_box_point(direction)
 else:
 point = mob_or_point
 result = np.zeros(self.dim)
 result[: len(point)] = point
 return result

def put_start_and_end_on(self, start, end):
 curr_start, curr_end = self.get_start_and_end()
 if (curr_start == curr_end).all():
 self.set_points_by_ends(start, end, self.path_arc)
 return super().put_start_and_end_on(start, end)

def get_vector(self):
 return self.get_end() - self.get_start()

def get_unit_vector(self):
 return normalize(self.get_vector())

def get_angle(self):
 return angle_of_vector(self.get_vector())

def get_projection(self, point):
 """
 Return projection of a point onto the line
 """
 unit_vect = self.get_unit_vector()
 start = self.get_start()
 return start + np.dot(point - start, unit_vect) * unit_vect

def get_slope(self):
 return np.tan(self.get_angle())

def set_angle(self, angle, about_point=None):
 if about_point is None:
 about_point = self.get_start()
 self.rotate(
 angle - self.get_angle(),
 about_point=about_point,
)
 return self

def set_length(self, length):

```

```

 self.scale(length / self.get_length())

class OpenGLDashedLine(OpenGLLine):
 def __init__(self, *args, dash_length=DEFAULT_DASH_LENGTH, dashed_ratio=0.5, **kwargs):
 self.dashed_ratio = dashed_ratio
 self.dash_length = dash_length
 super().__init__(*args, **kwargs)
 dashed_ratio = self.dashed_ratio
 num_dashes = self.calculate_num_dashes(dashed_ratio)
 dashes = OpenGLDashedVMobject(
 self,
 num_dashes=num_dashes,
 dashed_ratio=dashed_ratio,
)
 self.clear_points()
 self.add(*dashes)

 def calculate_num_dashes(self, dashed_ratio):
 return max(
 2,
 int(np.ceil((self.get_length() / self.dash_length) * dashed_ratio)),
)

 def get_start(self):
 if len(self.submobjects) > 0:
 return self.submobjects[0].get_start()
 else:
 return super().get_start()

 def get_end(self):
 if len(self.submobjects) > 0:
 return self.submobjects[-1].get_end()
 else:
 return super().get_end()

 def get_first_handle(self):
 return self.submobjects[0].points[1]

 def get_last_handle(self):
 return self.submobjects[-1].points[-2]

class OpenGLTangentLine(OpenGLLine):
 def __init__(self, vmob, alpha, length=1, d_alpha=1e-6, **kwargs):
 self.length = length
 self.d_alpha = d_alpha
 da = self.d_alpha
 a1 = clip(alpha - da, 0, 1)
 a2 = clip(alpha + da, 0, 1)
 super().__init__(vmob.pfp(a1), vmob.pfp(a2), **kwargs)
 self.scale(self.length / self.get_length())

class OpenGLElbow(OpenGLVMobject):
 def __init__(self, width=0.2, angle=0, **kwargs):
 self.angle = angle
 super().__init__(**kwargs)
 self.set_points_as_corners([UP, UP + RIGHT, RIGHT])
 self.set_width(width, about_point=ORIGIN)
 self.rotate(self.angle, about_point=ORIGIN)

class OpenGLArrow(OpenGLLine):
 def __init__(self,
 start=LEFT,
 end=RIGHT,
 path_arc=0,
 fill_color=GREY_A,
 fill_opacity=1,
 stroke_width=0,
 buff=MED_SMALL_BUFF,
 thickness=0.05,
 tip_width_ratio=5,
 tip_angle=PI / 3,
 max_tip_length_to_length_ratio=0.5,
 max_width_to_length_ratio=0.1,
 **kwargs,
):
 self.thickness = thickness
 self.tip_width_ratio = tip_width_ratio
 self.tip_angle = tip_angle
 self.max_tip_length_to_length_ratio = max_tip_length_to_length_ratio
 self.max_width_to_length_ratio = max_width_to_length_ratio

```

```

super().__init__(
 start=start,
 end=end,
 buff=buff,
 path_arc=path_arc,
 fill_color=fill_color,
 fill_opacity=fill_opacity,
 stroke_width=stroke_width,
 **kwargs,
)

def set_points_by_ends(self, start, end, buff=0, path_arc=0):
 # Find the right tip length and thickness
 vect = end - start
 length = max(np.linalg.norm(vect), 1e-8)
 thickness = self.thickness
 w_ratio = self.max_width_to_length_ratio / (thickness / length)
 if w_ratio < 1:
 thickness *= w_ratio

 tip_width = self.tip_width_ratio * thickness
 tip_length = tip_width / (2 * np.tan(self.tip_angle / 2))
 t_ratio = self.max_tip_length_to_length_ratio / (tip_length / length)
 if t_ratio < 1:
 tip_length *= t_ratio
 tip_width *= t_ratio

 # Find points for the stem
 if path_arc == 0:
 points1 = (length - tip_length) * np.array([RIGHT, 0.5 * RIGHT, ORIGIN])
 points1 += thickness * UP / 2
 points2 = points1[::-1] + thickness * DOWN
 else:
 # Solve for radius so that the tip-to-tail length matches |end - start|
 a = 2 * (1 - np.cos(path_arc))
 b = -2 * tip_length * np.sin(path_arc)
 c = tip_length**2 - length**2
 R = (-b + np.sqrt(b**2 - 4 * a * c)) / (2 * a)

 # Find arc points
 points1 = OpenGLArc.create_quadratic_bezier_points(path_arc)
 points2 = np.array(points1[::-1])
 points1 *= R + thickness / 2
 points2 *= R - thickness / 2
 if path_arc < 0:
 tip_length *= -1
 rot_T = rotation_matrix_transpose(PI / 2 - path_arc, OUT)
 for points in points1, points2:
 points[:] = np.dot(points, rot_T)
 points += R * DOWN

 self.set_points(points1)
 # Tip
 self.add_line_to(tip_width * UP / 2)
 self.add_line_to(tip_length * LEFT)
 self.tip_index = len(self.points) - 1
 self.add_line_to(tip_width * DOWN / 2)
 self.add_line_to(points2[0])
 # Close it out
 self.append_points(points2)
 self.add_line_to(points1[0])

 if length > 0:
 # Final correction
 super().scale(length / self.get_length())

 self.rotate(angle_of_vector(vect) - self.get_angle())
 self.rotate(
 PI / 2 - np.arccos(normalize(vect)[2]),
 axis=rotate_vector(self.get_unit_vector(), -PI / 2),
)
 self.shift(start - self.get_start())
 self.refresh_triangulation()

def reset_points_around_ends(self):
 self.set_points_by_ends(
 self.get_start(),
 self.get_end(),
 path_arc=self.path_arc,
)
 return self

def get_start(self):
 nppc = self.n_points_per_curve
 points = self.points
 return (points[0] + points[-nppc]) / 2

```

```

def get_end(self):
 return self.points[self.tip_index]

def put_start_and_end_on(self, start, end):
 self.set_points_by_ends(start, end, buff=0, path_arc=self.path_arc)
 return self

def scale(self, *args, **kwargs):
 super().scale(*args, **kwargs)
 self.reset_points_around_ends()
 return self

def set_thickness(self, thickness):
 self.thickness = thickness
 self.reset_points_around_ends()
 return self

def set_path_arc(self, path_arc):
 self.path_arc = path_arc
 self.reset_points_around_ends()
 return self

class OpenGLVector(OpenGLArrow):
 def __init__(self, direction=RIGHT, buff=0, **kwargs):
 self.buff = buff
 if len(direction) == 2:
 direction = np.hstack([direction, 0])
 super().__init__(ORIGIN, direction, buff=buff, **kwargs)

class OpenGLDoubleArrow(OpenGLArrow):
 def __init__(self, *args, **kwargs):
 super().__init__(*args, **kwargs)
 self.add_tip(at_start=True)

class OpenGLCubicBezier(OpenGLVMOBJECT):
 def __init__(self, a0, h0, h1, a1, **kwargs):
 super().__init__(**kwargs)
 self.add_cubic_bezier_curve(a0, h0, h1, a1)

class OpenGLPolygon(OpenGLVMOBJECT):
 def __init__(self, *vertices, **kwargs):
 self.vertices = vertices
 super().__init__(**kwargs)

 def init_points(self):
 verts = self.vertices
 self.set_points_as_corners([*verts, verts[0]])

 def get_vertices(self):
 return self.get_start_anchors()

 def round_corners(self, radius=0.5):
 vertices = self.get_vertices()
 arcs = []
 for v1, v2, v3 in adjacent_n_tuples(vertices, 3):
 vect1 = v2 - v1
 vect2 = v3 - v2
 unit_vect1 = normalize(vect1)
 unit_vect2 = normalize(vect2)
 angle = angle_between_vectors(vect1, vect2)
 # Negative radius gives concave curves
 angle *= np.sign(radius)
 # Distance between vertex and start of the arc
 cut_off_length = radius * np.tan(angle / 2)
 # Determines counterclockwise vs. clockwise
 sign = np.sign(np.cross(vect1, vect2)[2])
 arc = OpenGLArcBetweenPoints(
 v2 - unit_vect1 * cut_off_length,
 v2 + unit_vect2 * cut_off_length,
 angle=sign * angle,
 n_components=2,
)
 arcs.append(arc)

 self.clear_points()
 # To ensure that we loop through starting with last
 arcs = [arcs[-1], *arcs[:-1]]
 for arc1, arc2 in adjacent_pairs(arcs):
 self.append_points(arc1.points)
 line = OpenGLLine(arc1.get_end(), arc2.get_start())
 # Make sure anchors are evenly distributed
 len_ratio = line.get_length() / arc1.get_arc_length()
 line.insert_n_curves(int(arc1.get_num_curves() * len_ratio))

```

```

 self.append_points(line.points)
 return self

class OpenGLRegularPolygon(OpenGLPolygon):
 def __init__(self, n=6, start_angle=None, **kwargs):
 self.start_angle = start_angle
 if self.start_angle is None:
 if n % 2 == 0:
 self.start_angle = 0
 else:
 self.start_angle = 90 * DEGREES
 start_vect = rotate_vector(RIGHT, self.start_angle)
 vertices = compass_directions(n, start_vect)
 super().__init__(*vertices, **kwargs)

class OpenGLTriangle(OpenGLRegularPolygon):
 def __init__(self, **kwargs):
 super().__init__(n=3, **kwargs)

class OpenGLArrowTip(OpenGLTriangle):
 def __init__(
 self,
 fill_opacity=1,
 fill_color=WHITE,
 stroke_width=0,
 width=DEFAULT_ARROW_TIP_WIDTH,
 length=DEFAULT_ARROW_TIP_LENGTH,
 angle=0,
 **kwargs,
):
 super().__init__(
 start_angle=0,
 fill_opacity=fill_opacity,
 fill_color=fill_color,
 stroke_width=stroke_width,
 **kwargs,
)
 self.set_width(width, stretch=True)
 self.set_height(length, stretch=True)

 def get_base(self):
 return self.point_from_proportion(0.5)

 def get_tip_point(self):
 return self.points[0]

 def get_vector(self):
 return self.get_tip_point() - self.get_base()

 def get_angle(self):
 return angle_of_vector(self.get_vector())

 def get_length(self):
 return np.linalg.norm(self.get_vector())

class OpenGLRectangle(OpenGLPolygon):
 def __init__(self, color=WHITE, width=4.0, height=2.0, **kwargs):
 super().__init__(UR, UL, DL, DR, color=color, **kwargs)

 self.set_width(width, stretch=True)
 self.set_height(height, stretch=True)

class OpenGLSquare(OpenGLRectangle):
 def __init__(self, side_length=2.0, **kwargs):
 self.side_length = side_length

 super().__init__(height=side_length, width=side_length, **kwargs)

class OpenGLRoundedRectangle(OpenGLRectangle):
 def __init__(self, corner_radius=0.5, **kwargs):
 self.corner_radius = corner_radius
 super().__init__(**kwargs)
 self.round_corners(self.corner_radius)

```

# scale.py

```
from __future__ import annotations

import math
from typing import TYPE_CHECKING, Any, Iterable

import numpy as np

__all__ = ["LogBase", "LinearBase"]

from manim.mobject.text.numbers import Integer

if TYPE_CHECKING:
 from manim.mobject.mobject import Mobject

class _ScaleBase:
 """Scale baseclass for graphing/functions.

 Parameters

 custom_labels
 Whether to create custom labels when plotted on a :class:`~.NumberLine`.
 """

 def __init__(self, custom_labels: bool = False):
 self.custom_labels = custom_labels

 def function(self, value: float) -> float:
 """The function that will be used to scale the values.

 Parameters

 value
 The number/``np.ndarray`` to be scaled.

 Returns

 float
 The value after it has undergone the scaling.

 Raises

 NotImplementedError
 Must be subclassed.
 """
 raise NotImplementedError

 def inverse_function(self, value: float) -> float:
 """The inverse of ``function``. Used for plotting on a particular axis.

 Raises

 NotImplementedError
 Must be subclassed.
 """
 raise NotImplementedError

 def get_custom_labels(
 self,
 val_range: Iterable[float],
) -> Iterable[Mobject]:
 """Custom instructions for generating labels along an axis.

 Parameters

 val_range
 The position of labels. Also used for defining the content of the labels.

 Returns

 Dict
 A list consisting of the labels.
 Can be passed to :meth:`~.NumberLine.add_labels()` along with ``val_range``.

 Raises

 NotImplementedError
 Can be subclassed, optional.
 """
 raise NotImplementedError
```

```

class LinearBase(_ScaleBase):
 def __init__(self, scale_factor: float = 1.0):
 """The default scaling class.

 Parameters

 scale_factor
 The slope of the linear function, by default 1.0
 """

 super().__init__()
 self.scale_factor = scale_factor

 def function(self, value: float) -> float:
 """Multiplies the value by the scale factor.

 Parameters

 value
 Value to be multiplied by the scale factor.
 """
 return self.scale_factor * value

 def inverse_function(self, value: float) -> float:
 """Inverse of function. Divides the value by the scale factor.

 Parameters

 value
 value to be divided by the scale factor.
 """
 return value / self.scale_factor

class LogBase(_ScaleBase):
 def __init__(self, base: float = 10, custom_labels: bool = True):
 """Scale for logarithmic graphs/functions.

 Parameters

 base
 The base of the log, by default 10.
 custom_labels
 For use with :class:`~.Axes`:
 Whether or not to include ``LaTeX`` axis labels, by default True.

 Examples

 .. code-block:: python

 func = ParametricFunction(lambda x: x, scaling=LogBase(base=2))

 """
 super().__init__()
 self.base = base
 self.custom_labels = custom_labels

 def function(self, value: float) -> float:
 """Scales the value to fit it to a logarithmic scale.``self.function(5)==10**5``"""

 return self.base**value

 def inverse_function(self, value: float) -> float:
 """Inverse of ``function``. The value must be greater than 0"""

 if isinstance(value, np.ndarray):
 condition = value.any() <= 0
 func = lambda value, base: np.log(value) / np.log(base)
 else:
 condition = value <= 0
 func = math.log

 if condition:
 raise ValueError(
 "log(0) is undefined. Make sure the value is in the domain of the function"
)
 value = func(value, self.base)
 return value

 def get_custom_labels(
 self,
 val_range: Iterable[float],
 unit_decimal_places: int = 0,
 **base_config: dict[str, Any],
) -> list[Mobject]:
 """Produces custom :class:`~.Integer` labels in the form of ``10^2``.

 Parameters

```

```

val_range
 The iterable of values used to create the labels. Determines the exponent.
unit_decimal_places
 The number of decimal places to include in the exponent
base_config
 Additional arguments to be passed to :class:`~.Integer`.
"""

uses `format` syntax to control the number of decimal places.
tex_labels = [
 Integer(
 self.base,
 unit="^{%s}" % (f"{self.inverse_function(i)}:{unit_decimal_places}f"),
 **base_config,
)
 for i in val_range
]
return tex_labels
```

# polyhedra.py

```
"""General polyhedral class and platonic solids."""
from __future__ import annotations
from typing import TYPE_CHECKING
import numpy as np

from manim.mobject.geometry.polygram import Polygon
from manim.mobject.graph import Graph
from manim.mobject.three_d.three_dimensions import Dot3D
from manim.mobject.types.vectorized_mobject import VGroup

if TYPE_CHECKING:
 from manim.mobject.mobject import Mobject

__all__ = ["Polyhedron", "Tetrahedron", "Octahedron", "Icosahedron", "Dodecahedron"]

class Polyhedron(VGroup):
 """An abstract polyhedra class.

 In this implementation, polyhedra are defined with a list of vertex coordinates in space, and a list of faces. This implementation mirrors that of a standard polyhedral data format (OFF, object file format).

 Parameters

 vertex_coords
 A list of coordinates of the corresponding vertices in the polyhedron. Each coordinate will correspond to a vertex. The vertices are indexed with the usual indexing of Python.
 faces_list
 A list of faces. Each face is a sublist containing the indices of the vertices that form the corners of that face.
 faces_config
 Configuration for the polygons representing the faces of the polyhedron.
 graph_config
 Configuration for the graph containing the vertices and edges of the polyhedron.

 Examples

 To understand how to create a custom polyhedra, let's use the example of a rather simple one - a square pyramid.

 .. manim:: SquarePyramidScene
 :save_last_frame:

 class SquarePyramidScene(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 vertex_coords = [
 [1, 1, 0],
 [1, -1, 0],
 [-1, -1, 0],
 [-1, 1, 0],
 [0, 0, 2]
]
 faces_list = [
 [0, 1, 4],
 [1, 2, 4],
 [2, 3, 4],
 [3, 0, 4],
 [0, 1, 2, 3]
]
 pyramid = Polyhedron(vertex_coords, faces_list)
 self.add(pyramid)

 In defining the polyhedron above, we first defined the coordinates of the vertices. These are the corners of the square base, given as the first four coordinates in the vertex list, and the apex, the last coordinate in the list.

 Next, we define the faces of the polyhedron. The triangular surfaces of the pyramid are polygons with two adjacent vertices in the base and the vertex at the apex as corners. We thus define these surfaces in the first four elements of our face list. The last element defines the base of the pyramid.

 The graph and faces of polyhedra can also be accessed and modified directly, after instantiation. They are stored in the `graph` and `faces` attributes respectively.

 .. manim:: PolyhedronSubMobjects
 :save_last_frame:

 class PolyhedronSubMobjects(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 octahedron = Octahedron(edge_length = 3)
 octahedron.graph[0].set_color(RED)
 octahedron.faces[2].set_color(YELLOW)
 self.add(octahedron)
"""


```

```

def __init__(self,
 vertex_coords: list[list[float] | np.ndarray],
 faces_list: list[list[int]],
 faces_config: dict[str, str | int | float | bool] = {},
 graph_config: dict[str, str | int | float | bool] = {},
) :
 super().__init__()
 self.faces_config = dict(
 {"fill_opacity": 0.5, "shade_in_3d": True}, **faces_config
)
 self.graph_config = dict(
 {
 "vertex_type": Dot3D,
 "edge_config": {
 "stroke_opacity": 0, # I find that having the edges visible makes the polyhedra look weird
 },
 },
 **graph_config,
)
 self.vertex_coords = vertex_coords
 self.vertex_indices = list(range(len(self.vertex_coords)))
 self.layout = dict(enumerate(self.vertex_coords))
 self.faces_list = faces_list
 self.face_coords = [[self.layout[j] for j in i] for i in faces_list]
 self.edges = self.get_edges(self.faces_list)
 self.faces = self.create_faces(self.face_coords)
 self.graph = Graph(
 self.vertex_indices, self.edges, layout=self.layout, **self.graph_config
)
 self.add(self.faces, self.graph)
 self.add_updater(self.update_faces)

def get_edges(self, faces_list: list[list[int]]) -> list[tuple[int, int]]:
 """Creates list of cyclic pairwise tuples."""
 edges = []
 for face in faces_list:
 edges += zip(face, face[1:] + face[:1])
 return edges

def create_faces(self, face_coords: list[list[list | np.ndarray]],) -> VGroup:
 """Creates VGroup of faces from a list of face coordinates."""
 face_group = VGroup()
 for face in face_coords:
 face_group.add(Polygon(*face, **self.faces_config))
 return face_group

def update_faces(self, m: Mobject):
 face_coords = self.extract_face_coords()
 new_faces = self.create_faces(face_coords)
 self.faces.match_points(new_faces)

def extract_face_coords(self) -> list[list[np.ndarray]]:
 """Extracts the coordinates of the vertices in the graph.
 Used for updating faces.
 """
 new_vertex_coords = []
 for v in self.graph.vertices:
 new_vertex_coords.append(self.graph[v].get_center())
 layout = dict(enumerate(new_vertex_coords))
 return [[layout[j] for j in i] for i in self.faces_list]

class Tetrahedron(Polyhedron):
 """A tetrahedron, one of the five platonic solids. It has 4 faces, 6 edges, and 4 vertices.

 Parameters

 edge_length
 The length of an edge between any two vertices.

 Examples

 .. manim:: TetrahedronScene
 :save_last_frame:

 class TetrahedronScene(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 obj = Tetrahedron()
 self.add(obj)

 def __init__(self, edge_length: float = 1, **kwargs):
 unit = edge_length * np.sqrt(2) / 4
 super().__init__(
 vertex_coords=[
 np.array([unit, unit, unit]),
 np.array([unit, -unit, -unit]),

```

```

 np.array([-unit, unit, -unit]),
 np.array([-unit, -unit, unit]),
],
 faces_list=[[0, 1, 2], [3, 0, 2], [0, 1, 3], [3, 1, 2]],
 **kwargs,
)

class Octahedron(Polyhedron):
 """An octahedron, one of the five platonic solids. It has 8 faces, 12 edges and 6 vertices.

Parameters

edge_length
 The length of an edge between any two vertices.

Examples

.. manim:: OctahedronScene
:save_last_frame:

class OctahedronScene(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 obj = Octahedron()
 self.add(obj)
"""

def __init__(self, edge_length: float = 1, **kwargs):
 unit = edge_length * np.sqrt(2) / 2
 super().__init__(
 vertex_coords=[
 np.array([unit, 0, 0]),
 np.array([-unit, 0, 0]),
 np.array([0, unit, 0]),
 np.array([0, -unit, 0]),
 np.array([0, 0, unit]),
 np.array([0, 0, -unit]),
],
 faces_list=[
 [2, 4, 1],
 [0, 4, 2],
 [4, 3, 0],
 [1, 3, 4],
 [3, 5, 0],
 [1, 5, 3],
 [2, 5, 1],
 [0, 5, 2],
],
 **kwargs,
)

class Icosahedron(Polyhedron):
 """An icosahedron, one of the five platonic solids. It has 20 faces, 30 edges and 12 vertices

Parameters

edge_length
 The length of an edge between any two vertices.

Examples

.. manim:: IcosahedronScene
:save_last_frame:

class IcosahedronScene(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 obj = Icosahedron()
 self.add(obj)
"""

def __init__(self, edge_length: float = 1, **kwargs):
 unit_a = edge_length * ((1 + np.sqrt(5)) / 4)
 unit_b = edge_length * (1 / 2)
 super().__init__(
 vertex_coords=[
 np.array([0, unit_b, unit_a]),
 np.array([0, -unit_b, unit_a]),
 np.array([0, unit_b, -unit_a]),
 np.array([0, -unit_b, -unit_a]),
 np.array([unit_b, unit_a, 0]),
 np.array([-unit_b, unit_a, 0]),
 np.array([-unit_b, -unit_a, 0]),
 np.array([-unit_b, -unit_a, 0]),
 np.array([unit_a, 0, unit_b]),
 np.array([unit_a, 0, -unit_b]),
 np.array([-unit_a, 0, unit_b]),
 np.array([-unit_a, 0, -unit_b]),
],
 **kwargs,
)

```

```

faces_list=[
 [1, 8, 0],
 [1, 5, 7],
 [8, 5, 1],
 [7, 3, 5],
 [5, 9, 3],
 [8, 9, 5],
 [3, 2, 9],
 [9, 4, 2],
 [8, 4, 9],
 [0, 4, 8],
 [6, 4, 0],
 [6, 2, 4],
 [11, 2, 6],
 [3, 11, 2],
 [0, 6, 10],
 [10, 1, 0],
 [10, 7, 1],
 [11, 7, 3],
 [10, 11, 7],
 [10, 11, 6],
],
**kwargs,
)

class Dodecahedron(Polyhedron):
 """A dodecahedron, one of the five platonic solids. It has 12 faces, 30 edges and 20 vertices.

Parameters

edge_length
 The length of an edge between any two vertices.

Examples

.. manim:: DodecahedronScene
:save_last_frame:

class DodecahedronScene(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 obj = Dodecahedron()
 self.add(obj)
"""

def __init__(self, edge_length: float = 1, **kwargs):
 unit_a = edge_length * ((1 + np.sqrt(5)) / 4)
 unit_b = edge_length * ((3 + np.sqrt(5)) / 4)
 unit_c = edge_length * (1 / 2)
 super().__init__(
 vertex_coords=[
 np.array([unit_a, unit_a, unit_a]),
 np.array([unit_a, unit_a, -unit_a]),
 np.array([unit_a, -unit_a, unit_a]),
 np.array([unit_a, -unit_a, -unit_a]),
 np.array([-unit_a, unit_a, unit_a]),
 np.array([-unit_a, unit_a, -unit_a]),
 np.array([-unit_a, -unit_a, unit_a]),
 np.array([-unit_a, -unit_a, -unit_a]),
 np.array([0, unit_c, unit_b]),
 np.array([0, unit_c, -unit_b]),
 np.array([0, -unit_c, -unit_b]),
 np.array([0, -unit_c, unit_b]),
 np.array([unit_c, unit_b, 0]),
 np.array([-unit_c, unit_b, 0]),
 np.array([unit_c, -unit_b, 0]),
 np.array([-unit_c, -unit_b, 0]),
 np.array([unit_b, 0, unit_c]),
 np.array([-unit_b, 0, unit_c]),
 np.array([unit_b, 0, -unit_c]),
 np.array([-unit_b, 0, -unit_c]),
],
 faces_list=[
 [18, 16, 0, 12, 1],
 [3, 18, 16, 2, 14],
 [3, 10, 9, 1, 18],
 [1, 9, 5, 13, 12],
 [0, 8, 4, 13, 12],
 [2, 16, 0, 8, 11],
 [4, 17, 6, 11, 8],
 [17, 19, 5, 13, 4],
 [19, 7, 15, 6, 17],
 [6, 15, 14, 2, 11],
 [19, 5, 9, 10, 7],
 [7, 10, 3, 14, 15],
],
**kwargs,
)

```

# shader.py

```
from __future__ import annotations

import re
import textwrap
from pathlib import Path

import moderngl
import numpy as np

from .. import config
from ..utils import opengl
from ..utils.simple_functions import get_parameters

SHADER_FOLDER = Path(__file__).parent / "shaders"
shader_program_cache: dict = {}
file_path_to_code_map: dict = {}

__all__ = [
 "Object3D",
 "Mesh",
 "Shader",
 "FullScreenQuad",
]
]

def get_shader_code_from_file(file_path: Path) -> str:
 if file_path in file_path_to_code_map:
 return file_path_to_code_map[file_path]
 source = file_path.read_text()
 include_lines = re.findall(
 r"\#include (?P<include_path>.*\.glsl)$",
 source,
 flags=re.MULTILINE,
)
 for match in include_lines:
 include_path = match.group("include_path")
 included_code = get_shader_code_from_file(
 file_path.parent / include_path,
)
 source = source.replace(match.group(0), included_code)
 file_path_to_code_map[file_path] = source
 return source

def filter_attributes(unfiltered_attributes, attributes):
 # Construct attributes for only those needed by the shader.
 filtered_attributes_dtype = []
 for i, dtype_name in enumerate(unfiltered_attributes.dtype.names):
 if dtype_name in attributes:
 filtered_attributes_dtype.append(
 (
 dtype_name,
 unfiltered_attributes.dtype[i].subdtype[0].str,
 unfiltered_attributes.dtype[i].shape,
),
)
 filtered_attributes = np.zeros(
 unfiltered_attributes[unfiltered_attributes.dtype.names[0]].shape[0],
 dtype=filtered_attributes_dtype,
)
 for dtype_name in unfiltered_attributes.dtype.names:
 if dtype_name in attributes:
 filtered_attributes[dtype_name] = unfiltered_attributes[dtype_name]
 return filtered_attributes

class Object3D:
 def __init__(self, *children):
 self.model_matrix = np.eye(4)
 self.normal_matrix = np.eye(4)
 self.children = []
 self.parent = None
 self.add(*children)
 self.init_updaters()

 # TODO: Use path_func.
 def interpolate(self, start, end, alpha, _):
 self.model_matrix = (1 - alpha) * start.model_matrix + alpha * end.model_matrix
```

```

 self.normal_matrix = (
 1 - alpha
) * start.normal_matrix + alpha * end.normal_matrix

def single_copy(self):
 copy = Object3D()
 copy.model_matrix = self.model_matrix.copy()
 copy.normal_matrix = self.normal_matrix.copy()
 return copy

def copy(self):
 node_to_copy = {}

 bfs = [self]
 while bfs:
 node = bfs.pop(0)
 bfs.extend(node.children)

 node_copy = node.single_copy()
 node_to_copy[node] = node_copy

 # Add the copy to the copy of the parent.
 if node.parent is not None and node is not self:
 node_to_copy[node.parent].add(node_copy)
 return node_to_copy[self]

def add(self, *children):
 for child in children:
 if child.parent is not None:
 raise Exception(
 "Attempt to add child that's already added to another Object3D",
)
 self.remove(*children, current_children_only=False)
 self.children.extend(children)
 for child in children:
 child.parent = self

def remove(self, *children, current_children_only=True):
 if current_children_only:
 for child in children:
 if child.parent != self:
 raise Exception(
 "Attempt to remove child that isn't added to this Object3D",
)
 self.children = list(filter(lambda child: child not in children, self.children))
 for child in children:
 child.parent = None

def get_position(self):
 return self.model_matrix[:, 3][:3]

def set_position(self, position):
 self.model_matrix[:, 3][:3] = position
 return self

def get_meshes(self):
 dfs = [self]
 while dfs:
 parent = dfs.pop()
 if isinstance(parent, Mesh):
 yield parent
 dfs.extend(parent.children)

def get_family(self):
 dfs = [self]
 while dfs:
 parent = dfs.pop()
 yield parent
 dfs.extend(parent.children)

def align_data_and_family(self, _):
 pass

def hierarchical_model_matrix(self):
 if self.parent is None:
 return self.model_matrix

 model_matrices = [self.model_matrix]
 current_object = self
 while current_object.parent is not None:
 model_matrices.append(current_object.parent.model_matrix)
 current_object = current_object.parent
 return np.linalg.multi_dot(list(reversed(model_matrices)))

def hierarchical_normal_matrix(self):
 if self.parent is None:
 return self.normal_matrix[:3, :3]

```

```

normal_matrices = [self.normal_matrix]
current_object = self
while current_object.parent is not None:
 normal_matrices.append(current_object.parent.model_matrix)
 current_object = current_object.parent
return np.linalg.multi_dot(list(reversed(normal_matrices)))[::3, ::3]

def init_updaters(self):
 self.time_based_updaters = []
 self.non_time_updaters = []
 self.has_updaters = False
 self.updating_suspended = False

def update(self, dt=0):
 if not self.has_updaters or self.updating_suspended:
 return self
 for updater in self.time_based_updaters:
 updater(self, dt)
 for updater in self.non_time_updaters:
 updater(self)
 return self

def get_time_based_updaters(self):
 return self.time_based_updaters

def has_time_based_updater(self):
 return len(self.time_based_updaters) > 0

def get_updaters(self):
 return self.time_based_updaters + self.non_time_updaters

def add_updater(self, update_function, index=None, call_updater=True):
 if "dt" in get_parameters(update_function):
 updater_list = self.time_based_updaters
 else:
 updater_list = self.non_time_updaters

 if index is None:
 updater_list.append(update_function)
 else:
 updater_list.insert(index, update_function)

 self.refresh_has_updater_status()
 if call_updater:
 self.update()
 return self

def remove_updater(self, update_function):
 for updater_list in [self.time_based_updaters, self.non_time_updaters]:
 while update_function in updater_list:
 updater_list.remove(update_function)
 self.refresh_has_updater_status()
 return self

def clear_updaters(self):
 self.time_based_updaters = []
 self.non_time_updaters = []
 self.refresh_has_updater_status()
 return self

def match_updaters(self, mobject):
 self.clear_updaters()
 for updater in mobject.get_updaters():
 self.add_updater(updater)
 return self

def suspend_updating(self):
 self.updating_suspended = True
 return self

def resume_updating(self, call_updater=True):
 self.updating_suspended = False
 if call_updater:
 self.update(dt=0)
 return self

def refresh_has_updater_status(self):
 self.has_updaters = len(self.get_updaters()) > 0
 return self

class Mesh(Object3D):
 def __init__(self,
 shader=None,
 attributes=None,

```

```

geometry=None,
material=None,
indices=None,
use_depth_test=True,
primitive=moderngl.TRIANGLES,
):
 super().__init__()
 if shader is not None and attributes is not None:
 self.shader = shader
 self.attributes = attributes
 self.indices = indices
 elif geometry is not None and material is not None:
 self.shader = material
 self.attributes = geometry.attributes
 self.indices = geometry.index
 else:
 raise Exception(
 "Mesh requires either attributes and a Shader or a Geometry and a "
 "Material",
)
 self.use_depth_test = use_depth_test
 self.primitive = primitive
 self.skip_render = False
 self.init_updaters()

def single_copy(self):
 copy = Mesh(
 attributes=self.attributes.copy(),
 shader=self.shader,
 indices=self.indices.copy() if self.indices is not None else None,
 use_depth_test=self.use_depth_test,
 primitive=self.primitive,
)
 copy.skip_render = self.skip_render
 copy.model_matrix = self.model_matrix.copy()
 copy.normal_matrix = self.normal_matrix.copy()
 # TODO: Copy updaters?
 return copy

def set_uniforms(self, renderer):
 self.shader.set_uniform(
 "u_model_matrix",
 opengl.matrix_to_shader_input(self.model_matrix),
)
 self.shader.set_uniform("u_view_matrix", renderer.camera.formatted_view_matrix)
 self.shader.set_uniform(
 "u_projection_matrix",
 renderer.camera.projection_matrix,
)

def render(self):
 if self.skip_render:
 return

 if self.use_depth_test:
 self.shader.context.enable(moderngl.DEPTH_TEST)
 else:
 self.shader.context.disable(moderngl.DEPTH_TEST)

 from moderngl import Attribute

 shader_attributes = []
 for k, v in self.shader.shader_program._members.items():
 if isinstance(v, Attribute):
 shader_attributes.append(k)
 shader_attributes = filter_attributes(self.attributes, shader_attributes)

 vertex_buffer_object = self.shader.context.buffer(shader_attributes.tobytes())
 if self.indices is None:
 index_buffer_object = None
 else:
 vert_index_data = self.indices.astype("i4").tobytes()
 if vert_index_data:
 index_buffer_object = self.shader.context.buffer(vert_index_data)
 else:
 index_buffer_object = None
 vertex_array_object = self.shader.context.simple_vertex_array(
 self.shader.shader_program,
 vertex_buffer_object,
 *shader_attributes.dtype.names,
 index_buffer=index_buffer_object,
)
 vertex_array_object.render(self.primitive)
 vertex_buffer_object.release()
 vertex_array_object.release()
 if index_buffer_object is not None:
 index_buffer_object.release()

```

```

class Shader:
 def __init__(self,
 context,
 name=None,
 source=None,
):
 global shader_program_cache
 self.context = context
 self.name = name

 # See if the program is cached.
 if (
 self.name in shader_program_cache
 and shader_program_cache[self.name].ctx == self.context
):
 self.shader_program = shader_program_cache[self.name]
 elif source is not None:
 # Generate the shader from inline code if it was passed.
 self.shader_program = context.program(**source)
 else:
 # Search for a file containing the shader.
 source_dict = {}
 source_dict_key = {
 "vert": "vertex_shader",
 "frag": "fragment_shader",
 "geom": "geometry_shader",
 }
 shader_folder = SHADER_FOLDER / name
 for shader_file in shader_folder.iterdir():
 shader_file_path = shader_folder / shader_file
 shader_source = get_shader_code_from_file(shader_file_path)
 source_dict[source_dict_key[shader_file_path.stem]] = shader_source
 self.shader_program = context.program(**source_dict)

 # Cache the shader.
 if name is not None and name not in shader_program_cache:
 shader_program_cache[self.name] = self.shader_program

 def set_uniform(self, name, value):
 try:
 self.shader_program[name] = value
 except KeyError:
 pass

class FullScreenQuad(Mesh):
 def __init__(self,
 context,
 fragment_shader_source=None,
 fragment_shader_name=None,
):
 if fragment_shader_source is None and fragment_shader_name is None:
 raise Exception("Must either pass shader name or shader source.")

 if fragment_shader_name is not None:
 # Use the name.
 shader_file_path = SHADER_FOLDER / f"{fragment_shader_name}.frag"
 fragment_shader_source = get_shader_code_from_file(shader_file_path)
 elif fragment_shader_source is not None:
 fragment_shader_source = textwrap.dedent(fragment_shader_source.lstrip())

 shader = Shader(
 context,
 source={
 "vertex_shader": """
#version 330
in vec4 in_vert;
uniform mat4 u_model_view_matrix;
uniform mat4 u_projection_matrix;
void main() {
 vec4 camera_space_vertex = u_model_view_matrix * in_vert;
 vec4 clip_space_vertex = u_projection_matrix * camera_space_vertex;
 gl_Position = clip_space_vertex;
}
""",
 "fragment_shader": fragment_shader_source,
 },
)
 attributes = np.zeros(6, dtype=[("in_vert", np.float32, (4,))])
 attributes["in_vert"] = np.array([
 [-config["frame_x_radius"], -config["frame_y_radius"], 0, 1],
 [-config["frame_x_radius"], config["frame_y_radius"], 0, 1],

```

```
[config["frame_x_radius"], config["frame_y_radius"], 0, 1],
[-config["frame_x_radius"], -config["frame_y_radius"], 0, 1],
[config["frame_x_radius"], -config["frame_y_radius"], 0, 1],
[config["frame_x_radius"], config["frame_y_radius"], 0, 1],
],
)
shader.set_uniform("u_model_view_matrix", opengl.view_matrix())
shader.set_uniform(
 "u_projection_matrix",
 opengl.orthographic_projection_matrix(),
)
super().__init__(shader, attributes)

def render(self):
 super().render()
```

# changing.py

```
"""Animation of a mobject boundary and tracing of points."""

from __future__ import annotations

__all__ = ["AnimatedBoundary", "TracedPath"]

from typing import Callable

from colour import Color

from manim._config import config
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.color import BLUE_B, BLUE_D, BLUE_E, GREY_BROWN, WHITE
from manim.utils.rate_functions import smooth

class AnimatedBoundary(VGroup):
 """Boundary of a :class:`VMobject` with animated color change.

 Examples

 .. manim:: AnimatedBoundaryExample

 class AnimatedBoundaryExample(Scene):
 def construct(self):
 text = Text("So shiny!")
 boundary = AnimatedBoundary(text, colors=[RED, GREEN, BLUE],
 cycle_rate=3)
 self.add(text, boundary)
 self.wait(2)

 """

 def __init__(self,
 vmobject,
 colors=[BLUE_D, BLUE_B, BLUE_E, GREY_BROWN],
 max_stroke_width=3,
 cycle_rate=0.5,
 back_and_forth=True,
 draw_rate_func=smooth,
 fade_rate_func=smooth,
 **kwargs,
):
 super().__init__(**kwargs)
 self.colors = colors
 self.max_stroke_width = max_stroke_width
 self.cycle_rate = cycle_rate
 self.back_and_forth = back_and_forth
 self.draw_rate_func = draw_rate_func
 self.fade_rate_func = fade_rate_func
 self.vmobject = vmobject
 self.boundary_copies = [
 vmobject.copy().set_style(stroke_width=0, fill_opacity=0) for x in range(2)
]
 self.add(*self.boundary_copies)
 self.total_time = 0
 self.add_updater(lambda m, dt: self.update_boundary_copies(dt))

 def update_boundary_copies(self, dt):
 # Not actual time, but something which passes at
 # an altered rate to make the implementation below
 # cleaner
 time = self.total_time * self.cycle_rate
 growing, fading = self.boundary_copies
 colors = self.colors
 msw = self.max_stroke_width
 vmobject = self.vmobject

 index = int(time % len(colors))
 alpha = time % 1
 draw_alpha = self.draw_rate_func(alpha)
 fade_alpha = self.fade_rate_func(alpha)

 if self.back_and_forth and int(time) % 2 == 1:
 bounds = (1 - draw_alpha, 1)
 else:
 bounds = (0, draw_alpha)
 self.full_family_become_partial(growing, vmobject, *bounds)
 growing.set_stroke(colors[index], width=msw)
```

```

if time >= 1:
 self.full_family_become_partial(fading, vmobject, 0, 1)
 fading.set_stroke(color=colors[index - 1], width=(1 - fade_alpha) * msw)

self.total_time += dt

def full_family_become_partial(self, mob1, mob2, a, b):
 family1 = mob1.family_members_with_points()
 family2 = mob2.family_members_with_points()
 for sm1, sm2 in zip(family1, family2):
 sm1.pointwise_become_partial(sm2, a, b)
 return self

class TracedPath(VMobject, metaclass=ConvertToOpenGL):
 """Traces the path of a point returned by a function call.

 Parameters

 traced_point_func
 The function to be traced.
 stroke_width
 The width of the trace.
 stroke_color
 The color of the trace.
 dissipating_time
 The time taken for the path to dissipate. Default set to ``None`` which disables dissipation.

 Examples

 .. manim:: TracedPathExample

 class TracedPathExample(Scene):
 def construct(self):
 circ = Circle(color=RED).shift(4*LEFT)
 dot = Dot(color=RED).move_to(circ.get_start())
 rolling_circle = VGroup(circ, dot)
 trace = TracedPath(circ.get_start)
 rolling_circle.add_updater(lambda m: m.rotate(-0.3))
 self.add(trace, rolling_circle)
 self.play(rolling_circle.animate.shift(8*RIGHT), run_time=4, rate_func=linear)

 .. manim:: DissipatingPathExample

 class DissipatingPathExample(Scene):
 def construct(self):
 a = Dot(RIGHT * 2)
 b = TracedPath(a.get_center, dissipating_time=0.5, stroke_opacity=[0, 1])
 self.add(a, b)
 self.play(a.animate(path_arc=PI / 4).shift(LEFT * 2))
 self.play(a.animate(path_arc=-PI / 4).shift(LEFT * 2))
 self.wait()

 """

def __init__(self,
 traced_point_func: Callable,
 stroke_width: float = 2,
 stroke_color: Color = WHITE,
 dissipating_time: float | None = None,
 **kwargs,
):
 super().__init__(stroke_color=stroke_color, stroke_width=stroke_width, **kwargs)
 self.traced_point_func = traced_point_func
 self.dissipating_time = dissipating_time
 self.time = 1 if self.dissipating_time else None
 self.add_updater(self.update_path)

def update_path(self, mob, dt):
 new_point = self.traced_point_func()
 if not self.has_points():
 self.start_new_path(new_point)
 self.add_line_to(new_point)
 if self.dissipating_time:
 self.time += dt
 if self.time - 1 > self.dissipating_time:
 nppcc = self.n_points_per_curve
 self.set_points(self.points[nppcc:])

```

## \_\_main\_\_.py

```
from __future__ import annotations
import sys
import click
import cloup

from . import __version__, cli_ctx_settings, console
from .cli.cfg.group import cfg
from .cli.default_group import DefaultGroup
from .cli.init.commands import init
from .cli.new.group import new
from .cli.plugins.commands import plugins
from .cli.render.commands import render
from .constants import EPILOG

def exit_early(ctx, param, value):
 if value:
 sys.exit()

console.print(f"Manim Community [green]v{__version__}[/green]\n")

@cloup.group(
 context_settings=cli_ctx_settings,
 cls=DefaultGroup,
 default="render",
 no_args_is_help=True,
 help="Animation engine for explanatory math videos.",
 epilog="See 'manim <command>' to read about a specific subcommand.\n\n"
 "Note: the subcommand 'manim render' is called if no other subcommand "
 "is specified. Run 'manim render --help' if you would like to know what the "
 f"'-ql' or '-p' flags do, for example.\n\n{EPILOG}",
)
@click.option(
 "--version",
 is_flag=True,
 help="Show version and exit.",
 callback=exit_early,
 is_eager=True,
 expose_value=False,
)
@click.pass_context
def main(ctx):
 """The entry point for manim."""
 pass

main.add_command(cfg)
main.add_command(plugins)
main.add_command(init)
main.add_command(new)
main.add_command(render)

if __name__ == "__main__":
 main()
```

# frames\_comparison.py

```
from __future__ import annotations

import functools
import inspect
from pathlib import Path
from typing import Callable

from _pytest.fixtures import FixtureRequest

from manim import Scene
from manim._config import tempconfig
from manim._config.utils import ManimConfig
from manim.camera.three_d_camera import ThreeDCamera
from manim.renderer.cairo_renderer import CairoRenderer
from manim.scene.three_d_scene import ThreeDScene

from .frames_testers import _ControlDataWriter, _FramesTester
from .test_class_makers import (
 DummySceneFileWriter,
 _make_scene_file_writer_class,
 _make_test_renderer_class,
 _make_test_scene_class,
)
SCENE_PARAMETER_NAME = "scene"
_TESTS_ROOT_DIR_PATH = Path(__file__).absolute().parents[2]
PATH_CONTROL_DATA = _TESTS_ROOT_DIR_PATH / Path("control_data", "graphical_units_data")

def frames_comparison(
 func=None,
 *,
 last_frame: bool = True,
 renderer_class=CairoRenderer,
 base_scene=Scene,
 **custom_config,
):
 """Compares the frames generated by the test with control frames previously registered.

 If there is no control frames for this test, the test will fail. To generate
 control frames for a given test, pass ``--set_test`` flag to pytest
 while running the test.

 Note that this decorator can be used with or without parentheses.

 Parameters

 last_frame
 whether the test should test the last frame, by default True.
 renderer_class
 The base renderer to use (OpenGLRenderer/CairoRenderer), by default CairoRenderer
 base_scene
 The base class for the scene (ThreeDScene, etc.), by default Scene

 .. warning::
 By default, last_frame is True, which means that only the last frame is tested.
 If the scene has a moving animation, then the test must set last_frame to False.
 """
 def decorator_maker(tested_scene_construct):
 if (
 SCENE_PARAMETER_NAME
 not in inspect.getfullargspec(tested_scene_construct).args
):
 raise Exception(
 f"Invalid graphical test function test function : must have '{SCENE_PARAMETER_NAME}' as one of the parameters."
)

 # Exclude "scene" from the argument list of the signature.
 old_sig = inspect.signature(
 functools.partial(tested_scene_construct, scene=None),
)

 if "__module_test__" not in tested_scene_construct.__globals__:
 raise Exception(
 "There is no module test name indicated for the graphical unit test. You have to declare __module_test__ in the test file."
)
 module_name = tested_scene_construct.__globals__.get("__module_test__")
 test_name = tested_scene_construct.__name__[len("test_") :]

 @functools.wraps(tested_scene_construct)
 # The "request" parameter is meant to be used as a fixture by pytest. See below.
 def wrapper(*args, request: FixtureRequest, tmp_path, **kwargs):
 # Wraps the test_function to a construct method, to "freeze" the eventual additional arguments (parametrizations fixtures).
 construct = functools.partial(tested_scene_construct, *args, **kwargs)

 # Kwargs contains the eventual parametrization arguments.
 # This modifies the test_name so that it is defined by the parametrization
 # arguments too.
 # Example: if "length" is parametrized from 0 to 20, the kwargs
 # will be once with {"length": 1}, etc.
 test_name_with_param = test_name + "_".join(
 f"_{str(tup[0])}_[{str(tup[1])}]" for tup in kwargs.items()
)

 config_tests = _config_test(last_frame)

 config_tests["text_dir"] = tmp_path
 config_tests["tex_dir"] = tmp_path

 if last_frame:
```

```

 config_tests["frame_rate"] = 1
 config_tests["dry_run"] = True

 setting_test = request.config.getoption("--set_test")
 try:
 test_file_path = tested_scene_construct._globals__["__file__"]
 except Exception:
 test_file_path = None
 real_test = _make_test_comparing_frames(
 file_path=_control_data_path(
 test_file_path,
 module_name,
 test_name_with_param,
 setting_test,
),
 base_scene=base_scene,
 construct=construct,
 renderer_class=renderer_class,
 is_set_test_data_test=setting_test,
 last_frame=last_frame,
 show_diff=request.config.getoption("--show_diff"),
 size_frame=(config_tests["pixel_height"], config_tests["pixel_width"]),
)

 # Isolate the config used for the test, to avoid modifying the global config during the test run.
 with tempconfig(**config_tests, **custom_config):
 real_test()

parameters = list(old_sig.parameters.values())
Adds "request" param into the signature of the wrapper, to use the associated pytest fixture.
This fixture is needed to have access to flags value and pytest's config. See above.
if "request" not in old_sig.parameters:
 parameters += [inspect.Parameter("request", inspect.Parameter.KEYWORD_ONLY)]
if "tmp_path" not in old_sig.parameters:
 parameters += [
 inspect.Parameter("tmp_path", inspect.Parameter.KEYWORD_ONLY),
]
new_sig = old_sig.replace(parameters=parameters)
wrapper.__signature__ = new_sig

Reach a bit into pytest internals to hoist the marks from our wrapped
function.
setattr(wrapper, "pytestmark", [])
new_marks = getattr(tested_scene_construct, "pytestmark", [])
wrapper.pytestmark = new_marks
return wrapper

Case where the decorator is called with and without parentheses.
If func is None, callable(None) returns False
if callable(func):
 return decorator_maker(func)
return decorator_maker

def _make_test_comparing_frames(
 file_path: Path,
 base_scene: type[Scene],
 construct: Callable[[Scene], None],
 renderer_class: type, # Renderer type, there is no superclass renderer yet
 is_set_test_data_test: bool,
 last_frame: bool,
 show_diff: bool,
 size_frame: tuple,
) -> Callable[[], None]:
 """Create the real pytest test that will fail if the frames mismatch.

 Parameters

 file_path
 The path of the control frames.
 base_scene
 The base scene class.
 construct
 The construct method (= the test function)
 renderer_class
 The renderer base class.
 show_diff
 whether to visually show_diff (see --show_diff)

 Returns

 Callable[[], None]
 The pytest test.
 """

 if is_set_test_data_test:
 frames_tester = _ControlDataWriter(file_path, size_frame=size_frame)
 else:
 frames_tester = _FramesTester(file_path, show_diff=show_diff)

 file_writer_class = (
 _make_scene_file_writer_class(frames_tester)
 if not last_frame
 else DummySceneFileWriter
)
 testRenderer = _make_test_renderer_class(renderer_class)

 def real_test():
 with frames_tester.testing():
 sceneTested = _make_test_scene_class(
 base_scene=base_scene,
 construct_test=construct,
 # NOTE this is really ugly but it's due to the very bad design of the two renderers.
 # If you pass a custom renderer to the Scene, the Camera class given as an argument in the Scene
 # is not passed to the renderer. See __init__ of Scene.
 # This potentially prevents OpenGL testing.
 test_renderer=testRenderer(file_writer_class=file_writer_class)

```

```

 if base_scene is not ThreeDScene
 else testRenderer(
 file_writer_class=file_writer_class,
 camera_class=ThreeDCamera,
), # testRenderer(file_writer_class=file_writer_class),
)
scene_tested = sceneTested(skip_animations=True)
scene_tested.render()
if last_frame:
 frames_tester.check_frame(-1, scene_tested.renderer.get_frame())

return real_test

def _control_data_path(
 test_file_path: str | None, module_name: str, test_name: str, setting_test: bool
) -> Path:
 if test_file_path is None:
 # For some reason, path to test file containing @frames_comparison could not
 # be determined. Use local directory instead.
 test_file_path = __file__

 path = Path(test_file_path).absolute().parent / "control_data" / module_name

 if setting_test:
 # Create the directory if not existing.
 path.mkdir(exist_ok=True)
 if not setting_test and not path.exists():
 raise Exception(f"The control frames directory can't be found in {path}")
 path = (path / test_name).with_suffix(".npz")
 if not setting_test and not path.is_file():
 raise Exception(
 f"The control frame for the test {test_name} cannot be found in {path.parent}. "
 "Make sure you generated the control frames first."
)
 return path

def _config_test(last_frame: bool) -> ManimConfig:
 return ManimConfig().digest_file(
 str(
 Path(__file__).parent
 / (
 "config_graphical_tests_monoframe.cfg"
 if last_frame
 else "config_graphical_tests_multiframes.cfg"
),
),
)

```

## dot\_cloud.py

```
from __future__ import annotations
__all__ = ["TrueDot", "DotCloud"]

import numpy as np

from manim.constants import ORIGIN, RIGHT, UP
from manim.mobject.opengl.opengl_point_cloud_mobject import OpenGLMobject
from manim.utils.color import YELLOW

class DotCloud(OpenGLMobject):
 def __init__(self, color=YELLOW, stroke_width=2.0, radius=2.0, density=10, **kwargs):
 self.radius = radius
 self.epsilon = 1.0 / density
 super().__init__(stroke_width=stroke_width, density=density, color=color, **kwargs)

 def init_points(self):
 self.points = np.array([
 r * (np.cos(theta) * RIGHT + np.sin(theta) * UP)
 for r in np.arange(self.epsilon, self.radius, self.epsilon)
 # Num is equal to int(stop - start) / (step + 1) reformulated.
 for theta in np.linspace(
 0,
 2 * np.pi,
 num=int(2 * np.pi * (r + self.epsilon) / self.epsilon),
)
],
 dtype=np.float32,
)

 def make_3d(self, gloss=0.5, shadow=0.2):
 self.set_gloss(gloss)
 self.set_shadow(shadow)
 self.apply_depth_test()
 return self

class TrueDot(DotCloud):
 def __init__(self, center=ORIGIN, stroke_width=2.0, **kwargs):
 self.radius = stroke_width
 super().__init__(points=[center], stroke_width=stroke_width, **kwargs)
```

# space\_ops.py

```
"""Utility functions for two- and three-dimensional vectors."""

from __future__ import annotations

__all__ = [
 "quaternion_mult",
 "quaternion_from_angle_axis",
 "angle_axis_from_quaternion",
 "quaternion_conjugate",
 "rotate_vector",
 "thick_diagonal",
 "rotation_matrix",
 "rotation_about_z",
 "z_to_vector",
 "angle_of_vector",
 "angle_between_vectors",
 "normalize",
 "get_unit_normal",
 "compass_directions",
 "regular_vertices",
 "complex_to_R3",
 "R3_to_complex",
 "complex_func_to_R3_func",
 "center_of_mass",
 "midpoint",
 "find_intersection",
 "line_intersection",
 "get_winding_number",
 "shoelace",
 "shoelace_direction",
 "cross2d",
 "earclip_triangulation",
 "cartesian_to_spherical",
 "spherical_to_cartesian",
 "perpendicular_bisector",
]
import itertools as it
import math
from typing import Sequence

import numpy as np
from mapbox_earcut import triangulate_float32 as earcut
from scipy.spatial.transform import Rotation

from .. import config
from ..constants import DOWN, OUT, PI, RIGHT, TAU, UP, RendererType
from ..utils.iterables import adjacent_pairs

def norm_squared(v: float) -> float:
 return np.dot(v, v)

Quaternions
TODO, implement quaternion type

def quaternion_mult(
 *quats: Sequence[float],
) -> np.ndarray | list[float | np.ndarray]:
 """Gets the Hamilton product of the quaternions provided.
 For more information, check `this Wikipedia page
 <https://en.wikipedia.org/wiki/Quaternion>`__.

 Returns

 Union[np.ndarray, List[Union[float, np.ndarray]]]
 Returns a list of product of two quaternions.
 """
 if config.renderer == RendererType.OPENGL:
 if len(quats) == 0:
 return [1, 0, 0, 0]
 result = quats[0]
 for next_quat in quats[1:]:
 w1, x1, y1, z1 = result
 w2, x2, y2, z2 = next_quat
 result = [
 w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2,
 w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2,
 w1 * y2 + x1 * z2 + y1 * w2 - z1 * x2,
 w1 * z2 - x1 * y2 + y1 * x2 + z1 * w2,
]
 else:
 raise NotImplementedError("OpenGL renderer does not support quaternions yet")
```

```

 w1 * y2 + y1 * w2 + z1 * x2 - x1 * z2,
 w1 * z2 + z1 * w2 + x1 * y2 - y1 * x2,
]
 return result
elif config.renderer == RendererType.CAIRO:
 q1 = quats[0]
 q2 = quats[1]

 w1, x1, y1, z1 = q1
 w2, x2, y2, z2 = q2
 return np.array(
 [
 w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2,
 w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2,
 w1 * y2 + y1 * w2 + z1 * x2 - x1 * z2,
 w1 * z2 + z1 * w2 + x1 * y2 - y1 * x2,
],
)
)

def quaternion_from_angle_axis(
 angle: float,
 axis: np.ndarray,
 axis_normalized: bool = False,
) -> list[float]:
 """Gets a quaternion from an angle and an axis.
 For more information, check `this Wikipedia page
 <https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles>`__.

 Parameters

 angle
 The angle for the quaternion.
 axis
 The axis for the quaternion
 axis_normalized
 Checks whether the axis is normalized, by default False

 Returns

 List[float]
 Gives back a quaternion from the angle and axis
 """
 if config.renderer == RendererType.OPENGL:
 if not axis_normalized:
 axis = normalize(axis)
 return [math.cos(angle / 2), *(math.sin(angle / 2) * axis)]
 elif config.renderer == RendererType.CAIRO:
 return np.append(np.cos(angle / 2), np.sin(angle / 2) * normalize(axis))

def angle_axis_from_quaternion(quaternion: Sequence[float]) -> Sequence[float]:
 """Gets angle and axis from a quaternion.

 Parameters

 quaternion
 The quaternion from which we get the angle and axis.

 Returns

 Sequence[float]
 Gives the angle and axis
 """
 axis = normalize(quaternion[1:], fall_back=np.array([1, 0, 0]))
 angle = 2 * np.arccos(quaternion[0])
 if angle > TAU / 2:
 angle = TAU - angle
 return angle, axis

def quaternion_conjugate(quaternion: Sequence[float]) -> np.ndarray:
 """Used for finding the conjugate of the quaternion

 Parameters

 quaternion
 The quaternion for which you want to find the conjugate for.

 Returns

 np.ndarray
 The conjugate of the quaternion.
 """
 result = np.array(quaternion)
 result[1:] *= -1
 return result

```

```

def rotate_vector(
 vector: np.ndarray, angle: float, axis: np.ndarray = OUT
) -> np.ndarray:
 """Function for rotating a vector.

 Parameters

 vector
 The vector to be rotated.
 angle
 The angle to be rotated by.
 axis
 The axis to be rotated, by default OUT

 Returns

 np.ndarray
 The rotated vector with provided angle and axis.

 Raises

 ValueError
 If vector is not of dimension 2 or 3.
 """
 if len(vector) > 3:
 raise ValueError("Vector must have the correct dimensions.")
 if len(vector) == 2:
 vector = np.append(vector, 0)
 return rotation_matrix(angle, axis) @ vector

def thick_diagonal(dim: int, thickness=2) -> np.ndarray:
 row_indices = np.arange(dim).repeat(dim).reshape((dim, dim))
 col_indices = np.transpose(row_indices)
 return (np.abs(row_indices - col_indices) < thickness).astype("uint8")

def rotation_matrix_transpose_from_quaternion(quat: np.ndarray) -> list[np.ndarray]:
 """Converts the quaternion, quat, to an equivalent rotation matrix representation.
 For more information, check `this page
 <https://in.mathworks.com/help/driving/ref/quaternion.rotmat.html>`_.

 Parameters

 quat
 The quaternion which is to be converted.

 Returns

 List[np.ndarray]
 Gives back the Rotation matrix representation, returned as a 3-by-3
 matrix or 3-by-3-by-N multidimensional array.
 """
 quat_inv = quaternion_conjugate(quat)
 return [
 quaternion_mult(quat, [0, *basis], quat_inv)[1:]
 for basis in [
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1],
]
]

def rotation_matrix_from_quaternion(quat: np.ndarray) -> np.ndarray:
 return np.transpose(rotation_matrix_transpose_from_quaternion(quat))

def rotation_matrix_transpose(angle: float, axis: np.ndarray) -> np.ndarray:
 if all(np.array(axis)[:2] == np.zeros(2)):
 return rotation_about_z(angle * np.sign(axis[2])).T
 return rotation_matrix(angle, axis).T

def rotation_matrix(
 angle: float,
 axis: np.ndarray,
 homogeneous: bool = False,
) -> np.ndarray:
 """
 Rotation in R^3 about a specified axis of rotation.
 """
 inhomogeneous_rotation_matrix = Rotation.from_rotvec(
 angle * normalize(np.array(axis))

```

```

).as_matrix()
if not homogeneous:
 return inhomogeneous_rotation_matrix
else:
 rotation_matrix = np.eye(4)
 rotation_matrix[:3, :3] = inhomogeneous_rotation_matrix
 return rotation_matrix

def rotation_about_z(angle: float) -> np.ndarray:
 """Returns a rotation matrix for a given angle.

 Parameters

 angle
 Angle for the rotation matrix.

 Returns

 np.ndarray
 Gives back the rotated matrix.
 """
 c, s = math.cos(angle), math.sin(angle)
 return np.array(
 [
 [c, -s, 0],
 [s, c, 0],
 [0, 0, 1],
]
)

def z_to_vector(vector: np.ndarray) -> np.ndarray:
 """
 Returns some matrix in SO(3) which takes the z-axis to the
 (normalized) vector provided as an argument
 """
 axis_z = normalize(vector)
 axis_y = normalize(np.cross(axis_z, RIGHT))
 axis_x = np.cross(axis_y, axis_z)
 if np.linalg.norm(axis_y) == 0:
 # the vector passed just so happened to be in the x direction.
 axis_x = normalize(np.cross(UP, axis_z))
 axis_y = -np.cross(axis_x, axis_z)

 return np.array([axis_x, axis_y, axis_z]).T

def angle_of_vector(vector: Sequence[float] | np.ndarray) -> float:
 """Returns polar coordinate theta when vector is projected on xy plane.

 Parameters

 vector
 The vector to find the angle for.

 Returns

 float
 The angle of the vector projected.
 """
 if isinstance(vector, np.ndarray) and len(vector.shape) > 1:
 if vector.shape[0] < 2:
 raise ValueError("Vector must have the correct dimensions. (2, n)")
 c_vec = np.empty(vector.shape[1], dtype=np.complex128)
 c_vec.real = vector[0]
 c_vec.imag = vector[1]
 return np.angle(c_vec)
 return np.angle(complex(*vector[:2]))

def angle_between_vectors(v1: np.ndarray, v2: np.ndarray) -> float:
 """Returns the angle between two vectors.
 This angle will always be between 0 and pi

 Parameters

 v1
 The first vector.
 v2
 The second vector.

 Returns

 float
 The angle between the vectors.
 """

```

```

return 2 * np.arctan2(
 np.linalg.norm(normalize(v1) - normalize(v2)),
 np.linalg.norm(normalize(v1) + normalize(v2)),
)

def normalize(vect: np.ndarray | tuple[float], fall_back=None) -> np.ndarray:
 norm = np.linalg.norm(vect)
 if norm > 0:
 return np.array(vect) / norm
 else:
 return fall_back or np.zeros(len(vect))

def normalize_along_axis(array: np.ndarray, axis: np.ndarray) -> np.ndarray:
 """Normalizes an array with the provided axis.

 Parameters

 array
 The array which has to be normalized.
 axis
 The axis to be normalized to.

 Returns

 np.ndarray
 Array which has been normalized according to the axis.
 """
 norms = np.sqrt((array * array).sum(axis))
 norms[norms == 0] = 1
 buffed_norms = np.repeat(norms, array.shape[axis]).reshape(array.shape)
 array /= buffed_norms
 return array

def get_unit_normal(v1: np.ndarray, v2: np.ndarray, tol: float = 1e-6) -> np.ndarray:
 """Gets the unit normal of the vectors.

 Parameters

 v1
 The first vector.
 v2
 The second vector
 tol
 [description], by default 1e-6

 Returns

 np.ndarray
 The normal of the two vectors.
 """
 v1, v2 = (normalize(i) for i in (v1, v2))
 cp = np.cross(v1, v2)
 cp_norm = np.linalg.norm(cp)
 if cp_norm < tol:
 # Vectors align, so find a normal to them in the plane shared with the z-axis
 cp = np.cross(np.cross(v1, OUT), v1)
 cp_norm = np.linalg.norm(cp)
 if cp_norm < tol:
 return DOWN
 return normalize(cp)

 ##

def compass_directions(n: int = 4, start_vect: np.ndarray = RIGHT) -> np.ndarray:
 """Finds the cardinal directions using tau.

 Parameters

 n
 The amount to be rotated, by default 4
 start_vect
 The direction for the angle to start with, by default RIGHT

 Returns

 np.ndarray
 The angle which has been rotated.
 """
 angle = TAU / n
 return np.array([rotate_vector(start_vect, k * angle) for k in range(n)])

```

```

def regular_vertices(
 n: int, *, radius: float = 1, start_angle: float | None = None
) -> tuple[np.ndarray, float]:
 """Generates regularly spaced vertices around a circle centered at the origin.

 Parameters

 n
 The number of vertices
 radius
 The radius of the circle that the vertices are placed on.
 start_angle
 The angle the vertices start at.

 If unspecified, for even ``n`` values, ``0`` will be used.
 For odd ``n`` values, 90 degrees is used.

 Returns

 vertices : :class:`numpy.ndarray`
 The regularly spaced vertices.
 start_angle : :class:`float`
 The angle the vertices start at.
"""

if start_angle is None:
 if n % 2 == 0:
 start_angle = 0
 else:
 start_angle = TAU / 4

start_vector = rotate_vector(RIGHT * radius, start_angle)
vertices = compass_directions(n, start_vector)

return vertices, start_angle

def complex_to_R3(complex_num: complex) -> np.ndarray:
 return np.array((complex_num.real, complex_num.imag, 0))

def R3_to_complex(point: Sequence[float]) -> np.ndarray:
 return complex(*point[:2])

def complex_func_to_R3_func(complex_func):
 return lambda p: complex_to_R3(complex_func(R3_to_complex(p)))

def center_of_mass(points: Sequence[float]) -> np.ndarray:
 """Gets the center of mass of the points in space.

 Parameters

 points
 The points to find the center of mass from.

 Returns

 np.ndarray
 The center of mass of the points.
"""

 return np.average(points, 0, np.ones(len(points)))

def midpoint(
 point1: Sequence[float],
 point2: Sequence[float],
) -> float | np.ndarray:
 """Gets the midpoint of two points.

 Parameters

 point1
 The first point.
 point2
 The second point.

 Returns

 Union[float, np.ndarray]
 The midpoint of the points.
"""

 return center_of_mass([point1, point2])

```

```

def line_intersection(
 line1: Sequence[np.ndarray], line2: Sequence[np.ndarray]
) -> np.ndarray:
 """Returns the intersection point of two lines, each defined by
 a pair of distinct points lying on the line.

 Parameters

 line1
 A list of two points that determine the first line.
 line2
 A list of two points that determine the second line.

 Returns

 np.ndarray
 The intersection points of the two lines which are intersecting.

 Raises

 ValueError
 Error is produced if the two lines don't intersect with each other
 or if the coordinates don't lie on the xy-plane.
 """
 if any(np.array([line1, line2])[:, :, 2].reshape(-1)):
 # checks for z coordinates != 0
 raise ValueError("Coords must be in the xy-plane.")

 # algorithm from https://stackoverflow.com/a/42727584
 padded = (
 np.pad(np.array(i)[:, :2], ((0, 0), (0, 1)), constant_values=1)
 for i in (line1, line2)
)
 line1, line2 = (np.cross(*i) for i in padded)
 x, y, z = np.cross(line1, line2)

 if z == 0:
 raise ValueError(
 "The lines are parallel, there is no unique intersection point."
)

 return np.array([x / z, y / z, 0])

def find_intersection(
 p0s: Sequence[np.ndarray],
 v0s: Sequence[np.ndarray],
 p1s: Sequence[np.ndarray],
 v1s: Sequence[np.ndarray],
 threshold: float = 1e-5,
) -> Sequence[np.ndarray]:
 """
 Return the intersection of a line passing through p0 in direction v0
 with one passing through p1 in direction v1 (or array of intersections
 from arrays of such points/directions).
 For 3d values, it returns the point on the ray p0 + v0 * t closest to the
 ray p1 + v1 * t
 """
 # algorithm from https://en.wikipedia.org/wiki/Skew_lines#Nearest_points
 result = []

 for p0, v0, p1, v1 in zip(*[p0s, v0s, p1s, v1s]):
 normal = np.cross(v1, np.cross(v0, v1))
 denom = max(np.dot(v0, normal), threshold)
 result += [p0 + np.dot(p1 - p0, normal) / denom * v0]
 return result

def get_winding_number(points: Sequence[np.ndarray]) -> float:
 """Determine the number of times a polygon winds around the origin.

 The orientation is measured mathematically positively, i.e.,
 counterclockwise.

 Parameters

 points
 The vertices of the polygon being queried.

 Examples

 >>> from manim import Square, get_winding_number
 >>> polygon = Square()
 >>> get_winding_number(polygon.get_vertices())
 1.0
 >>> polygon.shift(2*UP)

```

```

Square
>>> get_winding_number(polygon.get_vertices())
0.0
"""
total_angle = 0
for p1, p2 in adjacent_pairs(points):
 d_angle = angle_of_vector(p2) - angle_of_vector(p1)
 d_angle = ((d_angle + PI) % TAU) - PI
 total_angle += d_angle
return total_angle / TAU

def shoelace(x_y: np.ndarray) -> float:
 """2D implementation of the shoelace formula.

 Returns

 :class:`float`
 Returns signed area.
 """
 x = x_y[:, 0]
 y = x_y[:, 1]
 return np.trapz(y, x)

def shoelace_direction(x_y: np.ndarray) -> str:
 """
 Uses the area determined by the shoelace method to determine whether
 the input set of points is directed clockwise or counterclockwise.

 Returns

 :class:`str`
 Either ``"CW"`` or ``"CCW"``.
 """
 area = shoelace(x_y)
 return "CW" if area > 0 else "CCW"

def cross2d(a, b):
 if len(a.shape) == 2:
 return a[:, 0] * b[:, 1] - a[:, 1] * b[:, 0]
 else:
 return a[0] * b[1] - b[0] * a[1]

def earclip_triangulation(verts: np.ndarray, ring_ends: list) -> list:
 """Returns a list of indices giving a triangulation
 of a polygon, potentially with holes.

 Parameters

 verts
 verts is a numpy array of points.
 ring_ends
 ring_ends is a list of indices indicating where
 the ends of new paths are.

 Returns

 list
 A list of indices giving a triangulation of a polygon.
 """
 # First, connect all the rings so that the polygon
 # with holes is instead treated as a (very convex)
 # polygon with one edge. Do this by drawing connections
 # between rings close to each other
 rings = [list(range(e0, e1)) for e0, e1 in zip([0, *ring_ends], ring_ends)]
 attached_rings = rings[:1]
 detached_rings = rings[1:]
 loop_connections = {}

 while detached_rings:
 i_range, j_range = (
 list(
 filter(
 # Ignore indices that are already being
 # used to draw some connection
 lambda i: i not in loop_connections,
 it.chain(*ring_group),
),
),
 for ring_group in (attached_rings, detached_rings)
)

 # Closest point on the attached rings to an estimated midpoint
 # of the detached rings

```

```

tmp_j_vert = midpoint(verts[j_range[0]], verts[j_range[len(j_range) // 2]])
i = min(i_range, key=lambda i: norm_squared(verts[i] - tmp_j_vert))
Closest point of the detached rings to the aforementioned
point of the attached rings
j = min(j_range, key=lambda j: norm_squared(verts[i] - verts[j]))
Recalculate i based on new j
i = min(i_range, key=lambda i: norm_squared(verts[i] - verts[j]))

Remember to connect the polygon at these points
loop_connections[i] = j
loop_connections[j] = i

Move the ring which j belongs to from the
attached list to the detached list
new_ring = next(filter(lambda ring: ring[0] <= j < ring[-1], detached_rings))
detached_rings.remove(new_ring)
attached_rings.append(new_ring)

Setup linked list
after = []
end0 = 0
for endl in ring_ends:
 after.extend(range(end0 + 1, endl))
 after.append(end0)
 end0 = endl

Find an ordering of indices walking around the polygon
indices = []
i = 0
for _ in range(len(verts) + len(ring_ends) - 1):
 # starting = False
 if i in loop_connections:
 j = loop_connections[i]
 indices.extend([i, j])
 i = after[j]
 else:
 indices.append(i)
 i = after[i]
 if i == 0:
 break

meta_indices = earcut(verts[indices, :2], [len(indices)])
return [indices[mi] for mi in meta_indices]

def cartesian_to_spherical(vec: Sequence[float]) -> np.ndarray:
 """Returns an array of numbers corresponding to each
 polar coordinate value (distance, phi, theta).

 Parameters

 vec
 A numpy array ``[x, y, z]``.
 """
 norm = np.linalg.norm(vec)
 if norm == 0:
 return 0, 0, 0
 r = norm
 phi = np.arccos(vec[2] / r)
 theta = np.arctan2(vec[1], vec[0])
 return np.array([r, theta, phi])

def spherical_to_cartesian(spherical: Sequence[float]) -> np.ndarray:
 """Returns a numpy array ``[x, y, z]`` based on the spherical
 coordinates given.

 Parameters

 spherical
 A list of three floats that correspond to the following:
 r - The distance between the point and the origin.
 theta - The azimuthal angle of the point to the positive x-axis.
 phi - The vertical angle of the point to the positive z-axis.
 """
 r, theta, phi = spherical
 return np.array([
 [
 r * np.cos(theta) * np.sin(phi),
 r * np.sin(theta) * np.sin(phi),
 r * np.cos(phi),
],
],
)

```

```
def perpendicular_bisector(
 line: Sequence[np.ndarray],
 norm_vector=OUT,
) -> Sequence[np.ndarray]:
 """Returns a list of two points that correspond
 to the ends of the perpendicular bisector of the
 two points given.

 Parameters

 line
 a list of two numpy array points (corresponding
 to the ends of a line).
 norm_vector
 the vector perpendicular to both the line given
 and the perpendicular bisector.

 Returns

 list
 A list of two numpy array points that correspond
 to the ends of the perpendicular bisector
 """
 p1 = line[0]
 p2 = line[1]
 direction = np.cross(p1 - p2, norm_vector)
 m = midpoint(p1, p2)
 return [m + direction, m - direction]
```

## coordinate systems.py

```

from __future__ import annotations

__all__ = [
 "CoordinateSystem",
 "Axes",
 "ThreeDAxes",
 "NumberPlane",
 "PolarPlane",
 "ComplexPlane",
]

import fractions as fr
import numbers
from typing import TYPE_CHECKING, Any, Callable, Iterable, Sequence

import numpy as np
from colour import Color

from manim import config
from manim.constants import *
from manim.mobject.geometry.arc import Circle, Dot
from manim.mobject.geometry.line import Arrow, DashedLine, Line
from manim.mobject.geometry.polygram import Polygon, Rectangle, RegularPolygon
from manim.mobject.graphing.functions import ImplicitFunction, ParametricFunction
from manim.mobject.number_line import NumberLine
from manim.mobject.graphing.scale import LinearBase
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl.opengl_surface import OpenGLSurface
from manim.mobject.text.tex_mobject import MathTex
from manim.mobject.three_d.three_dimensions import Surface
from manim.mobject.types.vectorized_mobject import (
 VDict,
 VectorizedPoint,
 VGroup,
 VMobject,
)
from manim.utils.color import (
 BLACK,
 BLUE,
 BLUE_D,
 GREEN,
 WHITE,
 YELLOW,
 color_gradient,
 invert_color,
)
from manim.utils.config_ops import merge_dicts_recursively, update_dict_recursively
from manim.utils.simple_functions import binary_search
from manim.utils.space_ops import angle_of_vector

if TYPE_CHECKING:
 from manim.mobject.mobject import Mobject

class CoordinateSystem:
 """Abstract base class for Axes and NumberPlane.

 Examples

 .. manim:: CoordSysExample
 :save_last_frame:

 class CoordSysExample(Scene):
 def construct(self):
 # the location of the ticks depends on the x_range and y_range.
 grid = Axes(
 x_range=[0, 1, 0.05], # step size determines num_decimal_places.
 y_range=[0, 1, 0.05],
 x_length=9,
 y_length=5.5,
 axis_config={
 "numbers_to_include": np.arange(0, 1 + 0.1, 0.1),
 "font_size": 24,
 },
 tips=False,
)

 # Labels for the x-axis and y-axis.
 y_label = grid.get_y_axis_label("y", edge=LEFT, direction=LEFT, buff=0.4)
 x_label = grid.get_x_axis_label("x")
 grid_labels = VGroup(x_label, y_label)

 graphs = VGroup()
 for n in np.arange(1, 20 + 0.5, 0.5):
 graphs += grid.plot(lambda x: x ** n, color=WHITE)
 graphs += grid.plot(
 lambda x: x ** (1 / n), color=WHITE, use_smoothing=False
)

 # Extra lines and labels for point (1,1)
 graphs += grid.get_horizontal_line(grid.c2p(1, 1, 0), color=BLUE)
 graphs += grid.get_vertical_line(grid.c2p(1, 1, 0), color=BLUE)
 graphs += Dot(point=grid.c2p(1, 1, 0), color=YELLOW)
 graphs += Tex("(1,1)").scale(0.75).next_to(grid.c2p(1, 1, 0))
 title = Title(
 "# spaces between braces to prevent SyntaxError
 r'Graphs of \$y=x^{ { 1 } \over { n } }\$ and \$y=x^n\$ (n=1,2,3,...,20)\$',
 include_underline=False,
 font_size=40,
)

 self.add(title, graphs, grid, grid_labels)
 """

 def __init__(self,
 x_range=None,
 y_range=None,
 x_length=None,
 y_length=None,
 dimension=2,
):
 self.dimension = dimension

 default_step = 1
 if x_range is None:

```

```

x_range = [
 round(-config["frame_x_radius"]),
 round(config["frame_x_radius"]),
 default_step,
]
elif len(x_range) == 2:
 x_range = [*x_range, default_step]

if y_range is None:
 y_range = [
 round(-config["frame_y_radius"]),
 round(config["frame_y_radius"]),
 default_step,
]
elif len(y_range) == 2:
 y_range = [*y_range, default_step]

self.x_range = x_range
self.y_range = y_range
self.x_length = x_length
self.y_length = y_length
self.num_sampled_graph_points_per_tick = 10

def coords_to_point(self, *coords):
 raise NotImplementedError()

def point_to_coords(self, point):
 raise NotImplementedError()

def polar_to_point(self, radius: float, azimuth: float) -> np.ndarray:
 """Gets a point from polar coordinates.

 Parameters

 radius
 The coordinate radius (:math:`r`).
 azimuth
 The coordinate azimuth (:math:`\theta`).

 Returns

 numpy.ndarray
 The point.

 Examples

 .. manim:: PolarToPointExample
 :ref_classes: PolarPlane Vector
 :save_last_frame:

 class PolarToPointExample(Scene):
 def construct(self):
 polarplane_pi = PolarPlane(azimuth_units="PI radians", size=6)
 polarpoint_vector = Vector(polarplane_pi.polar_to_point(3, PI/4))
 self.add(polarplane_pi)
 self.add(polarpoint_vector)

 """
 return self.coords_to_point(radius * np.cos(azimuth), radius * np.sin(azimuth))

def point_to_polar(self, point: np.ndarray) -> tuple[float, float]:
 """Gets polar coordinates from a point.

 Parameters

 point
 The point.

 Returns

 Tuple[:class:`float`, :class:`float`]
 The coordinate radius (:math:`r`) and the coordinate azimuth (:math:`\theta`).

 """
 x, y = self.point_to_coords(point)
 return np.sqrt(x**2 + y**2), np.arctan2(y, x)

def c2p(self, *coords):
 """Abbreviation for :meth:`coords_to_point`"""
 return self.coords_to_point(*coords)

def p2c(self, point):
 """Abbreviation for :meth:`point_to_coords`"""
 return self.point_to_coords(point)

def pr2pt(self, radius: float, azimuth: float) -> np.ndarray:
 """Abbreviation for :meth:`polar_to_point`"""
 return self.polar_to_point(radius, azimuth)

def pt2pr(self, point: np.ndarray) -> tuple[float, float]:
 """Abbreviation for :meth:`point_to_polar`"""
 return self.point_to_polar(point)

def get_axes(self):
 raise NotImplementedError()

def get_axis(self, index):
 return self.get_axes()[index]

def get_origin(self) -> np.ndarray:
 """Gets the origin of :class:`~.Axes`.

 Returns

 np.ndarray
 The center point.

 """
 return self.coords_to_point(0, 0)

def get_x_axis(self):
 return self.get_axis(0)

def get_y_axis(self):
 return self.get_axis(1)

def get_z_axis(self):
 return self.get_axis(2)

def get_x_unit_size(self):
 return self.get_x_axis().get_unit_size()

def get_y_unit_size(self):
 return self.get_y_axis().get_unit_size()

def get_x_axis_label(
 self,

```

```

label: float | str | Mobject,
edge: Sequence[float] = UR,
direction: Sequence[float] = UR,
buff: float = SMALL_BUFF,
**kwargs,
) -> Mobject:
 """Generate an x-axis label.

 Parameters

 label
 The label. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 edge
 The edge of the x-axis to which the label will be added, by default ``UR``.
 direction
 Allows for further positioning of the label from an edge, by default ``UR``.
 buff
 The distance of the label from the line.

 Returns

 :class:`~.Mobject`
 The positioned label.

 Examples

 .. manim:: GetXAxisLabelExample
 :save_last_frame:

 class GetXAxisLabelExample(Scene):
 def construct(self):
 ax = Axes(x_range=(0, 8), y_range=(0, 5), x_length=8, y_length=5)
 x_label = ax.get_x_axis_label(
 Tex("$$-values").scale(0.65), edge=DOWN, direction=DOWN, buff=0.5
)
 self.add(ax, x_label)

 return self._get_axis_label(
 label, self.get_x_axis(), edge, direction, buff=buff, **kwargs
)

 def get_y_axis_label(
 self,
 label: float | str | Mobject,
 edge: Sequence[float] = UR,
 direction: Sequence[float] = UP * 0.5 + RIGHT,
 buff: float = SMALL_BUFF,
 **kwargs,
):
 """Generate a y-axis label.

 Parameters

 label
 The label. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 edge
 The edge of the x-axis to which the label will be added, by default ``UR``.
 direction
 Allows for further positioning of the label from an edge, by default ``UR``.
 buff
 The distance of the label from the line.

 Returns

 :class:`~.Mobject`
 The positioned label.

 Examples

 .. manim:: GetYAxisLabelExample
 :save_last_frame:

 class GetYAxisLabelExample(Scene):
 def construct(self):
 ax = Axes(x_range=(0, 8), y_range=(0, 5), x_length=8, y_length=5)
 y_label = ax.get_y_axis_label(
 Tex("$$-values").scale(0.65).rotate(90 * DEGREES),
 edge=LEFT,
 direction=LEFT,
 buff=0.3,
)
 self.add(ax, y_label)

 return self._get_axis_label(
 label, self.get_y_axis(), edge, direction, buff=buff, **kwargs
)

 def _get_axis_label(
 self,
 label: float | str | Mobject,
 axis: Mobject,
 edge: Sequence[float],
 direction: Sequence[float],
 buff: float = SMALL_BUFF,
) -> Mobject:
 """Gets the label for an axis.

 Parameters

 label
 The label. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 axis
 The axis to which the label will be added.
 edge
 The edge of the axes to which the label will be added. ``RIGHT`` adds to the right side of the axis
 direction
 Allows for further positioning of the label.
 buff
 The distance of the label from the line.

 Returns

 :class:`~.Mobject`
 The positioned label along the given axis.

 """
 label = self.x_axis._create_label_tex(label)
 label.next_to(axis.get_edge_center(edge), direction=direction, buff=buff)
 label.shift_onto_screen(buff=MED_SMALL_BUFF)
 return label

 def get_axis_labels(self):
 raise NotImplementedError()

```

```

def add_coordinates(
 self,
 *axes_numbers: (Iterable[float] | None | dict[float, str | float | Mobject]),
 **kwargs,
):
 """Adds labels to the axes. Use ``Axes.coordinate_labels`` to access the coordinates after creation.

 Parameters

 axes_numbers
 The numbers to be added to the axes. Use ``None`` to represent an axis with default labels.

 Examples

 .. code-block:: python

 ax = ThreeDAxes()
 x_labels = range(-4, 5)
 z_labels = range(-4, 4, 2)
 ax.add_coordinates(x_labels, None, z_labels) # default y labels, custom x & z labels
 ax.add_coordinates(x_labels) # only x labels

 You can also specifically control the position and value of the labels using a dict.

 .. code-block:: python

 ax = Axes(x_range=[0, 7])
 x_pos = [x for x in range(1, 8)]

 # strings are automatically converted into a Tex mobobject.
 x_vals = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
 x_dict = dict(zip(x_pos, x_vals))
 ax.add_coordinates(x_dict)

 """
 self.coordinate_labels = VGroup()
 # if nothing is passed to axes_numbers, produce axes with default labelling
 if not axes_numbers:
 axes_numbers = [None for _ in range(self.dimension)]

 for axis, values in zip(self.axes, axes_numbers):
 if isinstance(values, dict):
 axis.add_labels(values, **kwargs)
 labels = axis.labels
 elif values is None and axis.scaling.custom_labels:
 tick_range = axis.get_tick_range()
 axis.add_labels(
 dict(zip(tick_range, axis.scaling.get_custom_labels(tick_range)))
)
 labels = axis.labels
 else:
 axis.add_numbers(values, **kwargs)
 labels = axis.labels
 self.coordinate_labels.add(labels)

 return self

def get_line_from_axis_to_point(
 self,
 index: int,
 point: Sequence[float],
 line_func: Line = DashedLine,
 line_config: dict | None = None,
 color: Color | None = None,
 stroke_width: float = 2,
) -> Line:
 """Returns a straight line from a given axis to a point in the scene.

 Parameters

 index
 Specifies the axis from which to draw the line. `0 = x_axis`, `1 = y_axis`
 point
 The point to which the line will be drawn.
 line_func
 The function of the :class:`~.Line` mobobject used to construct the line.
 line_config
 Optional arguments to pass to :attr:`line_func`.
 color
 The color of the line.
 stroke_width
 The stroke width of the line.

 Returns

 :class:`~.Line`
 The line from an axis to a point.

 .. seealso::
 :meth:`~.CoordinateSystem.get_vertical_line`
 :meth:`~.CoordinateSystem.get_horizontal_line`
 """

 line_config = line_config if line_config is not None else {}

 if color is None:
 color = VMobject().color

 line_config["color"] = color
 line_config["stroke_width"] = stroke_width

 axis = self.get_axis(index)
 line = line_func(axis.get_projection(point), point, **line_config)
 return line

def get_vertical_line(self, point: Sequence[float], **kwargs) -> Line:
 """A vertical line from the x-axis to a given point in the scene.

 Parameters

 point
 The point to which the vertical line will be drawn.
 kwargs
 Additional parameters to be passed to :class:`get_line_from_axis_to_point`.

 Returns

 :class:`Line`
 A vertical line from the x-axis to the point.

 Examples

 .. manim:: GetVerticalLineExample

```

```

:save_last_frame:

class GetVerticalLineExample(Scene):
 def construct(self):
 ax = Axes().add_coordinates()
 point = ax.coords_to_point(-3.5, 2)

 dot = Dot(point)
 line = ax.get_vertical_line(point, line_config={"dashed_ratio": 0.85})

 self.add(ax, line, dot)

"""

return self.get_line_from_axis_to_point(0, point, **kwargs)

def get_horizontal_line(self, point: Sequence[float], **kwargs) -> Line:
 """A horizontal line from the y-axis to a given point in the scene.

 Parameters

 point
 The point to which the horizontal line will be drawn.
 kwargs
 Additional parameters to be passed to :class:`get_line_from_axis_to_point`.

 Returns

 :class:`Line`
 A horizontal line from the y-axis to the point.

 Examples

 .. manim:: GetHorizontalLineExample
 :save_last_frame:

 class GetHorizontalLineExample(Scene):
 def construct(self):
 ax = Axes().add_coordinates()
 point = ax.c2p(-4, 1.5)

 dot = Dot(point)
 line = ax.get_horizontal_line(point, line_func=Line)

 self.add(ax, line, dot)

"""

return self.get_line_from_axis_to_point(1, point, **kwargs)

def get_lines_to_point(self, point: Sequence[float], **kwargs) -> VGroup:
 """Generate both horizontal and vertical lines from the axis to a point.

 Parameters

 point
 A point on the scene.
 kwargs
 Additional parameters to be passed to :meth:`get_line_from_axis_to_point`

 Returns

 :class:`~.VGroup`
 A :class:`~.VGroup` of the horizontal and vertical lines.

 Examples

 .. manim:: GetLinesToPointExample
 :save_last_frame:

 class GetLinesToPointExample(Scene):
 def construct(self):
 ax = Axes()
 circ = Circle(radius=0.5).move_to([-4, -1.5, 0])

 lines_1 = ax.get_lines_to_point(circ.get_right(), color=GREEN_B)
 lines_2 = ax.get_lines_to_point(circ.get_corner(DL), color=BLUE_B)
 self.add(ax, lines_1, lines_2, circ)

"""

return VGroup(
 self.get_horizontal_line(point, **kwargs),
 self.get_vertical_line(point, **kwargs),
)

graphing
def plot(
 self,
 function: Callable[[float], float],
 x_range: Sequence[float] | None = None,
 use_vectorized: bool = False,
 **kwargs,
):
 """Generates a curve based on a function.

 Parameters

 function
 The function used to construct the :class:`~.ParametricFunction`.
 x_range
 The range of the curve along the axes. ``x_range = [x_min, x_max, x_step]``.
 use_vectorized
 Whether to pass in the generated t value array to the function. Only use this if your function supports it.
 Output should be a numpy array of shape ``[y_0, y_1, ...]``
 kwargs
 Additional parameters to be passed to :class:`~.ParametricFunction`.

 Returns

 :class:`~.ParametricFunction`
 The plotted curve.

 Examples

 .. warning::
 This method may not produce accurate graphs since Manim currently relies on interpolation between evenly-spaced samples of the curve, instead of intelligent plotting.
 See the example below for some solutions to this problem.

 """

```

```

.. manim:: PlotExample
:save_last_frame:

class PlotExample(Scene):
 def construct(self):
 # construct the axes
 ax_1 = Axes(
 x_range=[0.001, 6],
 y_range=[-8, 2],
 x_length=5,
 y_length=3,
 tips=False,
)
 ax_2 = ax_1.copy()
 ax_3 = ax_1.copy()

 # position the axes
 ax_1.to_corner(UL)
 ax_2.to_corner(UR)
 ax_3.to_edge(DOWN)
 axes = VGroup(ax_1, ax_2, ax_3)

 # create the logarithmic curves
 def log_func(x):
 return np.log(x)

 # a curve without adjustments; poor interpolation.
 curve_1 = ax_1.plot(log_func, color=PURE_RED)

 # disabling interpolation makes the graph look choppy as not enough
 # inputs are available
 curve_2 = ax_2.plot(log_func, use_smoothing=False, color=ORANGE)

 # taking more inputs of the curve by specifying a step for the
 # x_range yields expected results, but increases rendering time.
 curve_3 = ax_3.plot(
 log_func, x_range=(0.001, 6, 0.001), color=PURE_GREEN
)

 curves = VGroup(curve_1, curve_2, curve_3)

 """
 self.add(axes, curves)

 """
 t_range = np.array(self.x_range, dtype=float)
 if x_range is not None:
 t_range[: len(x_range)] = x_range

 if x_range is None or len(x_range) < 3:
 # if t_range has a defined step size, increase the number of sample points per tick
 t_range[2] /= self.num_sampled_graph_points_per_tick
 # For axes, the third coordinate of x_range indicates
 # tick frequency. But for functions, it indicates a
 # sample frequency

 graph = ParametricFunction(
 lambda t: self.coords_to_point(t, function(t)),
 t_range=t_range,
 scaling=self.x_axis.scaling,
 use_vectorized=use_vectorized,
 **kwargs,
)
 graph.underlying_function = function
 return graph

def plot_implicit_curve(
 self,
 func: Callable,
 min_depth: int = 5,
 max_quads: int = 1500,
 **kwargs,
) -> ImplicitFunction:
 """Creates the curves of an implicit function.

 Parameters

 func
 The function to graph, in the form of $f(x, y) = 0$.
 min_depth
 The minimum depth of the function to calculate.
 max_quads
 The maximum number of quads to use.
 kwargs
 Additional parameters to pass into :class:`ImplicitFunction`.

 Examples

 .. manim:: ImplicitExample
 :save_last_frame:

 class ImplicitExample(Scene):
 def construct(self):
 a = ax.plot_implicit_curve(
 lambda x, y: y * (x - y) ** 2 - 4 * x - 8, color=BLUE
)
 self.add(ax, a)

 """
 x_scale = self.get_x_axis().scaling
 y_scale = self.get_y_axis().scaling
 graph = ImplicitFunction(
 func=(lambda x, y: func(x_scale.function(x), y_scale.function(y))),
 x_range=self.x_range[:2],
 y_range=self.y_range[:2],
 min_depth=min_depth,
 max_quads=max_quads,
 **kwargs,
)
 graph.stretch(self.get_x_unit_size(), 0, about_point=ORIGIN)
 .stretch(self.get_y_unit_size(), 1, about_point=ORIGIN)
 .shift(self.get_origin())
)
return graph

def plot_parametric_curve(
 self,
 function: Callable[[float], np.ndarray],
 use_vectorized: bool = False,
 **kwargs,
) -> ParametricFunction:
 """A parametric curve.

 Parameters

```

```

function
 A parametric function mapping a number to a point in the
 coordinate system.
use_vectorized
 Whether to pass in the generated t value array to the function. Only use this if your function supports it.
kwargs
 Any further keyword arguments are passed to :class:`.ParametricFunction`.

Example

.. manim:: ParametricCurveExample
 :save_last_frame:

 class ParametricCurveExample(Scene):
 def construct(self):
 ax = Axes()
 cardioid = ax.plot_parametric_curve(
 lambda t: np.array([
 np.exp(1) * np.cos(t) * (1 - np.cos(t)),
 np.exp(1) * np.sin(t) * (1 - np.cos(t)),
 0,
]),
 t_range=[0, 2 * PI],
 color="#0FF1CE",
)
 self.add(ax, cardioid)
 """
dim = self.dimension
graph = ParametricFunction(
 lambda t: self.coords_to_point(*function(t)[:dim]),
 use_vectorized=use_vectorized,
 **kwargs,
)
graph.underlying_function = function
return graph

def plot_polar_graph(
 self,
 r_func: Callable[[float], float],
 theta_range: Sequence[float] = [0, 2 * PI],
 **kwargs,
) -> ParametricFunction:
 """A polar graph.

Parameters

r_func
 The function r of theta.
theta_range
 The range of theta as ``theta_range = [theta_min, theta_max, theta_step]``.
kwargs
 Additional parameters passed to :class:`~.ParametricFunction`.

Examples

.. manim:: PolarGraphExample
 :ref_classes: PolarPlane
 :save_last_frame:

 class PolarGraphExample(Scene):
 def construct(self):
 plane = PolarPlane()
 r = lambda theta: 2 * np.sin(theta * 5)
 graph = plane.plot_polar_graph(r, [0, 2 * PI], color=ORANGE)
 self.add(plane, graph)
 """
graph = ParametricFunction(
 function=lambda th: self.pr2pt(r_func(th), th),
 t_range=theta_range,
 **kwargs,
)
graph.underlying_function = r_func
return graph

def plot_surface(
 self,
 function: Callable[[float], float],
 u_range: Sequence[float] | None = None,
 v_range: Sequence[float] | None = None,
 colorscale: Sequence[[color], float] | None = None,
 colorscale_axis: int = 2,
 **kwargs,
):
 """Generates a surface based on a function.

Parameters

function
 The function used to construct the :class:`~.Surface`.
u_range
 The range of the ``u`` variable: ``[u_min, u_max]``.
v_range
 The range of the ``v`` variable: ``[v_min, v_max]``.
colorscale
 Colors of the surface. Passing a list of colors will color the surface by z-value.
 Passing a list of tuples in the form ``[(color, pivot)]`` allows user-defined pivots
 where the color transitions.
colorscale_axis
 Defines the axis on which the colorscale is applied (0 = x, 1 = y, 2 = z), default
 is z-axis (2).
kwargs
 Additional parameters to be passed to :class:`~.Surface`.

Returns

:class:`~.Surface`
 The plotted surface.

Examples

.. manim:: PlotSurfaceExample
 :save_last_frame:

 class PlotSurfaceExample(ThreeDScene):
 def construct(self):
 resolution_fa = 16
 self.set_camera_orientation(phi=75 * DEGREES, theta=-60 * DEGREES)
 axes = ThreeDAxes(x_range=(-3, 3, 1), y_range=(-3, 3, 1), z_range=(-5, 5, 1))
 def param_trig(u, v):
 x = u
 y = v
 z = 2 * np.sin(x) + 2 * np.cos(y)
 return z

```

```

 trig_plane = axes.plot_surface(
 param_trig,
 resolution=(resolution_fa, resolution_fa),
 u_range = (-3, 3),
 v_range = (-3, 3),
 colorscale = [BLUE, GREEN, YELLOW, ORANGE, RED],
)
 self.add(axes, trig_plane)
 """
 if config.renderer == RendererType.CAIRO:
 surface = Surface(
 lambda u, v: self.c2p(u, v, function(u, v)),
 u_range=u_range,
 v_range=v_range,
 **kwargs,
)
 if colorscale:
 surface.set_fill_by_value(
 axes=self.copy(),
 colorscale=colorscale,
 axis=colorscale_axis,
)
 elif config.renderer == RendererType.OPENGL:
 surface = OpenGLSurface(
 lambda u, v: self.c2p(u, v, function(u, v)),
 u_range=u_range,
 v_range=v_range,
 axes=self.copy(),
 colorscale=colorscale,
 colorscale_axis=colorscale_axis,
 **kwargs,
)
)
 return surface

def input_to_graph_point(
 self,
 x: float,
 graph: ParametricFunction | VMobject,
) -> np.ndarray:
 """Returns the coordinates of the point on a ``graph`` corresponding to an ``x`` value.

 Parameters

 x
 The x-value of a point on the ``graph``.
 graph
 The :class:`~.ParametricFunction` on which the point lies.

 Returns

 :class:`np.ndarray`
 The coordinates of the point on the :attr:`graph` corresponding to the :attr:`x` value.

 Raises

 :exc:`ValueError`
 When the target x is not in the range of the line graph.

 Examples

 ... manim:: InputToGraphPointExample
 :save_last_frame:

 class InputToGraphPointExample(Scene):
 def construct(self):
 ax = Axes()
 curve = ax.plot(lambda x : np.cos(x))

 # move a square to PI on the cosine curve.
 position = ax.input_to_graph_point(x=PI, graph=curve)
 sq = Square(side_length=1, color=YELLOW).move_to(position)

 self.add(ax, curve, sq)
 """

 if hasattr(graph, "underlying_function"):
 return graph.function(x)
 else:
 alpha = binary_search(
 function=lambda a: self.point_to_coords(graph.point_from_proportion(a))[
 0
],
 target=x,
 lower_bound=0,
 upper_bound=1,
)
 if alpha is not None:
 return graph.point_from_proportion(alpha)
 else:
 raise ValueError(
 f"x={x} not located in the range of the graph ({[self.p2c(graph.get_start())[0]}, {self.p2c(graph.get_end())[0]}])",
)

def input_to_graph_coords(self, x: float, graph: ParametricFunction) -> tuple:
 """Returns a tuple of the axis relative coordinates of the point on the graph based on the x-value given.

 Examples

 ... code-block:: pycon

 >>> from manim import Axes
 >>> ax = Axes()
 >>> parabola = ax.plot(lambda x: x**2)
 >>> ax.input_to_graph_coords(x=3, graph=parabola)
 (3, 9)
 """
 return x, graph.underlying_function(x)

def i2gc(self, x: float, graph: ParametricFunction) -> tuple:
 """Alias for :meth:`input_to_graph_coords`."""
 return self.input_to_graph_coords(x, graph)

def i2gp(self, x: float, graph: ParametricFunction) -> np.ndarray:
 """Alias for :meth:`input_to_graph_point`."""
 return self.input_to_graph_point(x, graph)

def get_graph_label(
 self,
 graph: ParametricFunction,
 label: float | str | Mobject = "f(x)",
 x_val: float | None = None,
 direction: Sequence[float] = RIGHT,
 buff: float = MED_SMALL_BUFF,

```

```

color: Color | None = None,
dot: bool = False,
dot_config: dict | None = None,
) -> Mobject:
 """Creates a properly positioned label for the passed graph, with an optional dot.

Parameters

graph
 The curve.
label
 The label for the function's curve. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
x_val
 The x_value along the curve that positions the label.
direction
 The cartesian position, relative to the curve that the label will be at --> ``LEFT``, ``RIGHT``.
buff
 The distance between the curve and the label.
color
 The color of the label. Defaults to the color of the curve.
dot
 Whether to add a dot at the point on the graph.
dot_config
 Additional parameters to be passed to :class:`~.Dot`.

Returns

:class:`Mobject`
 The positioned label and :class:`~.Dot`, if applicable.

Examples

.. manim:: GetGraphLabelExample
 :save_last_frame:

 class GetGraphLabelExample(Scene):
 def construct(self):
 ax = Axes()
 sin = ax.plot(lambda x: np.sin(x), color=PURPLE_B)
 label = ax.get_graph_label(
 graph=sin,
 label=MathTex(r"\frac{\pi}{2}"),
 x_val=PI / 2,
 dot=True,
 direction=UR,
)
 self.add(ax, sin, label)

 if dot_config is None:
 dot_config = {}
 color = color or graph.get_color()
 label = self.x_axis._create_label_tex(label).set_color(color)

 if x_val is None:
 # Search from right to left
 for x in np.linspace(self.x_range[1], self.x_range[0], 100):
 point = self.input_to_graph_point(x, graph)
 if point[1] < config["frame_y_radius"]:
 break
 else:
 point = self.input_to_graph_point(x_val, graph)
 label.next_to(point, direction, buff=buff)
 label.shift_onto_screen()

 if dot:
 dot = Dot(point=point, **dot_config)
 label.add(dot)
 label.dot = dot
 return label

calculus

def get_riemann_rectangles(
 self,
 graph: ParametricFunction,
 x_range: Sequence[float] | None = None,
 dx: float | None = 0.1,
 input_sample_type: str = "left",
 stroke_width: float = 1,
 stroke_color: Color = BLACK,
 fill_opacity: float = 1,
 color: Iterable[Color] | Color = np.array((BLUE, GREEN)),
 show_signed_area: bool = True,
 bounded_graph: ParametricFunction = None,
 blend: bool = False,
 width_scale_factor: float = 1.001,
) -> VGroup:
 """Generates a :class:`~.VGroup` of the Riemann Rectangles for a given curve.

Parameters

graph
 The graph whose area will be approximated by Riemann rectangles.
x_range
 The minimum and maximum x-values of the rectangles. ``x_range = [x_min, x_max]``.
dx
 The change in x-value that separates each rectangle.
input_sample_type
 Can be any of ``"left"`` , ``"right"`` or ``"center"`` . Refers to where the sample point for the height of each Riemann Rectangle will be inside the segments of the partition.
stroke_width
 The stroke_width of the border of the rectangles.
stroke_color
 The color of the border of the rectangle.
fill_opacity
 The opacity of the rectangles.
color
 The colors of the rectangles. Creates a balanced gradient if multiple colors are passed.
show_signed_area
 Indicates negative area when the curve dips below the x-axis by inverting its color.
blend
 Sets the :attr:`stroke_color` to :attr:`fill_color`, blending the rectangles without clear separation.
bounded_graph
 If a secondary graph is specified, encloses the area between the two curves.
width_scale_factor
 The factor by which the width of the rectangles is scaled.

Returns

:class:`~.VGroup`
 A :class:`~.VGroup` containing the Riemann Rectangles.

```

```

Examples

.. manim:: GetRiemannRectanglesExample
:save_last_frame:

class GetRiemannRectanglesExample(Scene):
 def construct(self):
 ax = Axes(y_range=[-2, 10])
 quadratic = ax.plot(lambda x: 0.5 * x ** 2 - 0.5)

 # the rectangles are constructed from their top right corner.
 # passing an iterable to 'color' produces a gradient
 rects_right = ax.get_riemann_rectangles(
 quadratic,
 x_range=[-4, -3],
 dx=0.25,
 color=(TEAL, BLUE_B, DARK_BLUE),
 input_sample_type="right",
)

 # the colour of rectangles below the x-axis is inverted
 # due to show_signed_area
 rects_left = ax.get_riemann_rectangles(
 quadratic, x_range=[-1.5, 1.5], dx=0.15, color=YELLOW
)

 bounding_line = ax.plot(
 lambda x: 1.5 * x, color=BLUE_B, x_range=[3.3, 6]
)
 bounded_rects = ax.get_riemann_rectangles(
 bounding_line,
 bounded_graph=quadratic,
 dx=0.15,
 x_range=[4, 5],
 show_signed_area=False,
 color=(MAROON_A, RED_B, PURPLE_D),
)

 self.add(
 ax, bounding_line, quadratic, rects_right, rects_left, bounded_rects
)
"""

setting up x_range, overwrite user's third input
if x_range is None:
 if bounded_graph is None:
 x_range = [graph.t_min, graph.t_max]
 else:
 x_min = max(graph.t_min, bounded_graph.t_min)
 x_max = min(graph.t_max, bounded_graph.t_max)
 x_range = [x_min, x_max]

x_range = [*x_range[:2], dx]

rectangles = VGroup()
x_range = np.arange(*x_range)

allows passing a string to color the graph
if type(color) is str:
 colors = [color] * len(x_range)
else:
 colors = color_gradient(color, len(x_range))

for x, color in zip(x_range, colors):
 if input_sample_type == "left":
 sample_input = x
 elif input_sample_type == "right":
 sample_input = x + dx
 elif input_sample_type == "center":
 sample_input = x + 0.5 * dx
 else:
 raise ValueError("Invalid input sample type")
 graph_point = self.input_to_graph_point(sample_input, graph)

 if bounded_graph is None:
 y_point = self._origin_shift(self.y_range)
 else:
 y_point = bounded_graph.underlying_function(x)

 points = VGroup(
 *list(
 map(
 VectorizedPoint,
 [
 self.coords_to_point(x, y_point),
 self.coords_to_point(x + width_scale_factor * dx, y_point),
 graph_point,
],
),
)
)

 rect = Rectangle().replace(points, stretch=True)
 rectangles.add(rect)

 # checks if the rectangle is under the x-axis
 if self.p2c(graph_point)[1] < y_point and show_signed_area:
 color = invert_color(color)

 # blends rectangles smoothly
 if blend:
 stroke_color = color

 rect.set_style(
 fill_color=color,
 fill_opacity=fill_opacity,
 stroke_color=stroke_color,
 stroke_width=stroke_width,
)

return rectangles

def get_area(
 self,
 graph: ParametricFunction,
 x_range: tuple[float, float] | None = None,
 color: Color | Iterable[Color] = [BLUE, GREEN],
 opacity: float = 0.3,
 bounded_graph: ParametricFunction = None,
 **kwargs,
):
 """
 Returns a :class:`~.Polygon` representing the area under the graph passed.
 """

```

Parameters

```

graph
 The graph/curve for which the area needs to be gotten.
x_range
 The range of the minimum and maximum x-values of the area. ``x_range = [x_min, x_max]``.
color
 The color of the area. Creates a gradient if a list of colors is provided.
opacity
 The opacity of the area.
bounded_graph
 If a secondary :attr:`graph` is specified, encloses the area between the two curves.
kwargs
 Additional parameters passed to :class:`~.Polygon`.

>Returns

:class:`~.Polygon`
 The :class:`~.Polygon` representing the area.

>Raises

:exc:`ValueError`
 When x_ranges do not match (either area x_range, graph's x_range or bounded_graph's x_range).

>Examples

.. manim:: GetAreaExample
 :save_last_frame:

 class GetAreaExample(Scene):
 def construct(self):
 ax = Axes().add_coordinates()
 curve = ax.plot(lambda x: 2 * np.sin(x), color=DARK_BLUE)
 area = ax.get_area(
 curve,
 x_range=(PI / 2, 3 * PI / 2),
 color=(GREEN_B, GREEN_D),
 opacity=1,
)
 self.add(ax, curve, area)

 if x_range is None:
 a = graph.t_min
 b = graph.t_max
 else:
 a, b = x_range
 if bounded_graph is not None:
 if bounded_graph.t_min > b:
 raise ValueError(
 f"Ranges not matching: {bounded_graph.t_min} < {b}",
)
 if bounded_graph.t_max < a:
 raise ValueError(
 f"Ranges not matching: {bounded_graph.t_max} > {a}",
)
 a = max(a, bounded_graph.t_min)
 b = min(b, bounded_graph.t_max)

 if bounded_graph is None:
 points = (
 [self.c2p(a, graph.function(a))]
 + [p for p in graph.points if a <= self.p2c(p)[0] <= b]
 + [graph.function(b), self.c2p(b)]
)
 else:
 graph_points, bounded_graph_points = (
 [g.function(a)]
 + [p for p in g.points if a <= self.p2c(p)[0] <= b]
 + [g.function(b)]
 for g in (graph, bounded_graph)
)
 points = graph_points + bounded_graph_points[::-1]
 return Polygon(*points, **kwargs).set_opacity(opacity).set_color(color)

def angle_of_tangent(
 self,
 x: float,
 graph: ParametricFunction,
 dx: float = 1e-8,
) -> float:
 """Returns the angle to the x-axis of the tangent
 to the plotted curve at a particular x-value.

 Parameters

 x
 The x-value at which the tangent must touch the curve.
 graph
 The :class:`~.ParametricFunction` for which to calculate the tangent.
 dx
 The change in `x` used to determine the angle of the tangent to the curve.

 >Returns

 :class:`float`
 The angle of the tangent to the curve.

 >Examples

 .. code-block:: python

 ax = Axes()
 curve = ax.plot(lambda x: x**2)
 ax.angle_of_tangent(x=3, graph=curve)
 # 1.4056476493802699
 """

 p0 = np.array([*self.input_to_graph_coords(x, graph)])
 p1 = np.array([*self.input_to_graph_coords(x + dx, graph)])
 return angle_of_vector(p1 - p0)

def slope_of_tangent(self, x: float, graph: ParametricFunction, **kwargs) -> float:
 """Returns the slope of the tangent to the plotted curve
 at a particular x-value.

 Parameters

 x
 The x-value at which the tangent must touch the curve.
 graph
 The :class:`~.ParametricFunction` for which to calculate the tangent.

 >Returns

```

```

:class:`float`
 The slope of the tangent with the x axis.

Examples

.. code-block:: python

 ax = Axes()
 curve = ax.plot(lambda x: x**2)
 ax.slope_of_tangent(x=-2, graph=curve)
 # -3.5000000259052038
"""

return np.tan(self.angle_of_tangent(x, graph, **kwargs))

def plot_derivative_graph(
 self, graph: ParametricFunction, color: Color = GREEN, **kwargs
) -> ParametricFunction:
 """Returns the curve of the derivative of the passed graph.

Parameters

graph
 The graph for which the derivative will be found.
color
 The color of the derivative curve.
kwargs
 Any valid keyword argument of :class:`~.ParametricFunction`.

Returns

:class:`~.ParametricFunction`
 The curve of the derivative.

Examples

.. manim:: DerivativeGraphExample
 :save_last_frame:

 class DerivativeGraphExample(Scene):
 def construct(self):
 ax = NumberPlane(y_range=[-1, 7], background_line_style={"stroke_opacity": 0.4})

 curve_1 = ax.plot(lambda x: x ** 2, color=PURPLE_B)
 curve_2 = ax.plot_derivative_graph(curve_1)
 curves = VGroup(curve_1, curve_2)

 label_1 = ax.get_graph_label(curve_1, "x^2", x_val=-2, direction=DL)
 label_2 = ax.get_graph_label(curve_2, "2x", x_val=3, direction=RIGHT)
 labels = VGroup(label_1, label_2)

 self.add(ax, curves, labels)

 def deriv(x):
 return self.slope_of_tangent(x, graph)

 return self.plot(deriv, color=color, **kwargs)

def plot_antiderivative_graph(
 self,
 graph: ParametricFunction,
 y_intercept: float = 0,
 samples: int = 50,
 use_vectorized: bool = False,
 **kwargs,
):
 """Plots an antiderivative graph.

Parameters

graph
 The graph for which the antiderivative will be found.
y_intercept
 The y-value at which the graph intercepts the y-axis.
samples
 The number of points to take the area under the graph.
use_vectorized
 Whether to use the vectorized version of the antiderivative. This means
 to pass in the generated t value array to the function. Only use this if your function supports it.
 Output should be a numpy array of shape ``[y_0, y_1, ...]``.
kwargs
 Any valid keyword argument of :class:`~.ParametricFunction`.

Returns

:class:`~.ParametricFunction`
 The curve of the antiderivative.

.. note::
 This graph is plotted from the values of area under the reference graph.
 The result might not be ideal if the reference graph contains uncalculatable
 areas from x=0.

Examples

.. manim:: AntiderivativeExample
 :save_last_frame:

 class AntiderivativeExample(Scene):
 def construct(self):
 ax = Axes()
 graph1 = ax.plot(
 lambda x: (x ** 2 - 2) / 3,
 color=RED,
)
 graph2 = ax.plot_antiderivative_graph(graph1, color=BLUE)
 self.add(ax, graph1, graph2)

 def antideriv(x):
 x_vals = np.linspace(0, x, samples, axis=1 if use_vectorized else 0)
 f_vec = np.vectorize(graph.underlying_function)
 y_vals = f_vec(x_vals)
 return np.trapz(y_vals, x_vals) + y_intercept

 return self.plot(antideriv, use_vectorized=use_vectorized, **kwargs)

def get_secant_slope_group(
 self,
 x: float,
 graph: ParametricFunction,
 dx: float | None = None,
 dx_line_color: Color = YELLOW,

```

```

dy_line_color: Color | None = None,
dx_label: float | str | None = None,
dy_label: float | str | None = None,
include_secant_line: bool = True,
secant_line_color: Color = GREEN,
secant_line_length: float = 10,
) -> VGroup:
 """Creates two lines representing 'dx' and 'df', the labels for 'dx' and 'df', and
 the secant to the curve at a particular x-value.

 Parameters

 x
 The x-value at which the secant intersects the graph for the first time.
 graph
 The curve for which the secant will be found.
 dx
 The change in 'x' after which the secant exits.
 dx_line_color
 The color of the line that indicates the change in 'x'.
 dy_line_color
 The color of the line that indicates the change in 'y'. Defaults to the color of :attr:`graph`.
 dx_label
 The label for the 'dx' line. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 dy_label
 The label for the 'dy' line. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 include_secant_line
 Whether to include the secant line in the graph,
 or just the df/dx lines and labels.
 secant_line_color
 The color of the secant line.
 secant_line_length
 The length of the secant line.

 Returns

 :class:`~.VGroup`
 A group containing the elements: `dx_line`, `df_line`, and
 if applicable also :attr:`dx_label`, :attr:`df_label`, `secant_line`.

 Examples

 .. manim:: GetSecantSlopeGroupExample
 :save_last_frame:

 class GetSecantSlopeGroupExample(Scene):
 def construct(self):
 ax = Axes(y_range=[-1, 7])
 graph = ax.plot(lambda x: 1 / 4 * x ** 2, color=BLUE)
 slopes = ax.get_secant_slope_group(
 x=2.0,
 graph=graph,
 dx=1.0,
 dx_label=Tex("dx = 1.0"),
 dy_label="dy",
 dx_line_color=GREEN_B,
 secant_line_length=4,
 secant_line_color=RED_D,
)
 self.add(ax, graph, slopes)
 """
 group = VGroup()
 dx = dx or float(self.x_range[1] - self.x_range[0]) / 10
 dy_line_color = dy_line_color or graph.get_color()

 p1 = self.input_to_graph_point(x, graph)
 p2 = self.input_to_graph_point(x + dx, graph)
 interim_point = p2[0] * RIGHT + p1[1] * UP

 group.dx_line = Line(p1, interim_point, color=dx_line_color)
 group.df_line = Line(interim_point, p2, color=dy_line_color)
 group.add(group.dx_line, group.df_line)

 labels = VGroup()
 if dx_label is not None:
 group.dx_label = self.x_axis._create_label_tex(dx_label)
 labels.add(group.dx_label)
 group.add(group.dx_label)
 if dy_label is not None:
 group.df_label = self.x_axis._create_label_tex(dy_label)
 labels.add(group.df_label)
 group.add(group.df_label)

 if len(labels) > 0:
 max_width = 0.8 * group.dx_line.width
 max_height = 0.8 * group.df_line.height
 if labels.width > max_width:
 labels.width = max_width
 if labels.height > max_height:
 labels.height = max_height

 if dx_label is not None:
 group.dx_label.next_to(
 group.dx_line,
 np.sign(dx) * DOWN,
 buff=group.dx_label.height / 2,
)
 group.dx_label.set_color(group.dx_line.get_color())

 if dy_label is not None:
 group.df_label.next_to(
 group.df_line,
 np.sign(dx) * RIGHT,
 buff=group.df_label.height / 2,
)
 group.df_label.set_color(group.df_line.get_color())

 if include_secant_line:
 group.secant_line = Line(p1, p2, color=secant_line_color)
 group.secant_line.scale(
 secant_line_length / group.secant_line.get_length(),
)
 group.add(group.secant_line)
 return group

def get_vertical_lines_to_graph(
 self,
 graph: ParametricFunction,
 x_range: Sequence[float] | None = None,
 num_lines: int = 20,
 **kwargs,
) -> VGroup:
 """Obtains multiple lines from the x-axis to the curve.

```

```

Parameters

graph
 The graph along which the lines are placed.
x_range
 A list containing the lower and upper bounds of the lines: ``x_range = [x_min, x_max]``.
num_lines
 The number of evenly spaced lines.
kwargs
 Additional arguments to be passed to :meth:`~.CoordinateSystem.get_vertical_line`.

Returns

:class:`~.VGroup`
 The :class:`~.VGroup` of the evenly spaced lines.

Examples

.. manim:: GetVerticalLinesToGraph
 :save_last_frame:

 class GetVerticalLinesToGraph(Scene):
 def construct(self):
 ax = Axes(
 x_range=[0, 8.0, 1],
 y_range=[-1, 1, 0.2],
 axis_config={"font_size": 24},
).add_coordinates()

 curve = ax.plot(lambda x: np.sin(x) / np.e ** 2 * x)

 lines = ax.get_vertical_lines_to_graph(
 curve, x_range=[0, 4], num_lines=30, color=BLUE
)

 self.add(ax, curve, lines)
 ...

 x_range = x_range if x_range is not None else self.x_range

 return VGroup(
 *(self.get_vertical_line(self.i2gp(x, graph), **kwargs)
 for x in np.linspace(x_range[0], x_range[1], num_lines))
)
)

 def get_T_label(
 self,
 x_val: float,
 graph: ParametricFunction,
 label: float | str | Mobject | None = None,
 label_color: Color | None = None,
 triangle_size: float = MED_SMALL_BUFF,
 triangle_color: Color | None = WHITE,
 line_func: Line = Line,
 line_color: Color = YELLOW,
) -> VGroup:
 """Creates a labelled triangle marker with a vertical line from the x-axis
 to a curve at a given x-value.

 Parameters

 x_val
 The position along the curve at which the label, line and triangle will be constructed.
 graph
 The :class:`~.ParametricFunction` for which to construct the label.
 label
 The label of the vertical line and triangle.
 label_color
 The color of the label.
 triangle_size
 The size of the triangle.
 triangle_color
 The color of the triangle.
 line_func
 The function used to construct the vertical line.
 line_color
 The color of the vertical line.

 Returns

 :class:`~.VGroup`
 A :class:`~.VGroup` of the label, triangle and vertical line mobjects.

Examples

.. manim:: TLabelExample
 :save_last_frame:

 class TLabelExample(Scene):
 def construct(self):
 # defines the axes and linear function
 axes = Axes(x_range=[-1, 10], y_range=[-1, 10], x_length=9, y_length=6)
 func = axes.plot(lambda x: x, color=BLUE)
 # creates the T_label
 t_label = axes.get_T_label(x_val=4, graph=func, label=Tex("x-value"))
 self.add(axes, func, t_label)
 ...

 T_label_group = VGroup()
 triangle = RegularPolygon(n=3, start_angle=np.pi / 2, stroke_width=0).set_fill(
 color=triangle_color,
 opacity=1,
)
 triangle.height = triangle_size
 triangle.move_to(self.coords_to_point(x_val, 0), UP)
 if label is not None:
 t_label = self.x_axis.create_label_tex(label, color=label_color)
 t_label.next_to(triangle, DOWN)
 T_label_group.add(t_label)

 v_line = self.get_vertical_line(
 self.i2gp(x_val, graph),
 color=line_color,
 line_func=line_func,
)
 T_label_group.add(triangle, v_line)

 return T_label_group

class Axes(VGroup, CoordinateSystem, metaclass=ConvertToOpenGL):

```

```

"""Creates a set of axes.

Parameters

x_range
 The ``(x_min, x_max, x_step)`` values of the x-axis.
y_range
 The ``(y_min, y_max, y_step)`` values of the y-axis.
x_length
 The length of the x-axis.
y_length
 The length of the y-axis.
axis_config
 Arguments to be passed to :class:`~.NumberLine` that influences both axes.
x_axis_config
 Arguments to be passed to :class:`~.NumberLine` that influence the x-axis.
y_axis_config
 Arguments to be passed to :class:`~.NumberLine` that influence the y-axis.
tips
 Whether or not to include the tips on both axes.
kwargs
 Additional arguments to be passed to :class:`CoordinateSystem` and :class:`VGroup`.

Examples

.. manim:: LogScalingExample
 :save_last_frame:

 class LogScalingExample(Scene):
 def construct(self):
 ax = Axes(
 x_range=[0, 10, 1],
 y_range=[-2, 6, 1],
 tips=False,
 axis_config={"include_numbers": True},
 y_axis_config={"scaling": LogBase(custom_labels=True)},
)

 # x_min must be > 0 because log is undefined at 0.
 graph = ax.plot(lambda x: x ** 2, x_range=[0.001, 10], use_smoothing=False)
 self.add(ax, graph)

Styling arguments can be passed to the underlying :class:`.NumberLine` objects that represent the axes:
.. manim:: AxesWithDifferentTips
 :save_last_frame:

 class AxesWithDifferentTips(Scene):
 def construct(self):
 ax = Axes(axis_config={'tip_shape': StealthTip})
 self.add(ax)
 """

def __init__(self,
 x_range: Sequence[float] | None = None,
 y_range: Sequence[float] | None = None,
 x_length: float | None = round(config.frame_width) - 2,
 y_length: float | None = round(config.frame_height) - 2,
 axis_config: dict | None = None,
 x_axis_config: dict | None = None,
 y_axis_config: dict | None = None,
 tips: bool = True,
 **kwargs,
):
 VGroup.__init__(self, **kwargs)
 CoordinateSystem.__init__(self, x_range, y_range, x_length, y_length)

 self.axis_config = {
 "include_tip": tips,
 "numbers_to_exclude": [0],
 }
 self.x_axis_config = {}
 self.y_axis_config = {"rotation": 90 * DEGREES, "label_direction": LEFT}

 self._update_default_configs(
 (self.axis_config, self.x_axis_config, self.y_axis_config),
 (axis_config, x_axis_config, y_axis_config),
)

 self.x_axis_config = merge_dicts_recursively(
 self.axis_config,
 self.x_axis_config,
)
 self.y_axis_config = merge_dicts_recursively(
 self.axis_config,
 self.y_axis_config,
)

excluding the origin tick removes a tick at the 0-point of the axis
This is desired for LinearBase because the 0 point is always the x-axis
For non-LinearBase, the "0-point" does not have this quality, so it must be included.

i.e. with LogBase range [-2, 4]:
it would remove the "0" tick, which is actually 10^0,
not the lowest tick on the graph (which is 10^-2).

if self.x_axis_config.get("scaling") is None or isinstance(
 self.x_axis_config.get("scaling"), LinearBase
):
 self.x_axis_config["exclude_origin_tick"] = True
else:
 self.x_axis_config["exclude_origin_tick"] = False

if self.y_axis_config.get("scaling") is None or isinstance(
 self.y_axis_config.get("scaling"), LinearBase
):
 self.y_axis_config["exclude_origin_tick"] = True
else:
 self.y_axis_config["exclude_origin_tick"] = False

self.x_axis = self._create_axis(self.x_range, self.x_axis_config, self.x_length)
self.y_axis = self._create_axis(self.y_range, self.y_axis_config, self.y_length)

Add as a separate group in case various other
mobjects are added to self, as for example in
NumberPlane below
self.axes = VGroup(self.x_axis, self.y_axis)
self.add(*self.axes)

finds the middle-point on each axis
lines_center_point = [
 axis.scaling.function((axis.x_range[1] + axis.x_range[0]) / 2)
 for axis in self.axes
]

```

```

]

self.shift(-self.coords_to_point(*lines_center_point))

@staticmethod
def _update_default_configs(
 default_configs: tuple[dict[Any, Any]], passed_configs: tuple[dict[Any, Any]]
):
 """Takes in two tuples of dicts and return modifies the first such that values from
 ``passed_configs`` overwrite values in ``default_configs``. If a key does not exist
 in default_configs, it is added to the dict.

This method is useful for having defaults in a class and being able to overwrite
them with user-defined input.

Parameters

default_configs
 The dict that will be updated.
passed_configs
 The dict that will be used to update.

To create a tuple with one dictionary, add a comma after the element:

.. code-block:: python

 self._update_default_configs(
 (dict_1, (
 dict_2,
))
)
"""

for default_config, passed_config in zip(default_configs, passed_configs):
 if passed_config is not None:
 update_dict_recursively(default_config, passed_config)

def _create_axis(
 self,
 range_terms: Sequence[float],
 axis_config: dict,
 length: float,
) -> NumberLine:
 """Creates an axis and dynamically adjusts its position depending on where 0 is located on the line.

Parameters

range_terms
 The range of the the axis : ````(x_min, x_max, x_step)````.
axis_config
 Additional parameters that are passed to :class:`~.NumberLine`.
length
 The length of the axis.

Returns

:class:`NumberLine`
 Returns a number line based on ``range_terms``.

axis_config["length"] = length
axis = NumberLine(range_terms, **axis_config)

without the call to _origin_shift, graph does not exist when min > 0 or max < 0
shifts the axis so that 0 is centered
axis.shift(-axis.number_to_point(self._origin_shift([axis.x_min, axis.x_max])))
return axis

def coords_to_point(
 self, *coords: float | Sequence[float] | Sequence[Sequence[float]] | np.ndarray
) -> np.ndarray:
 """Accepts coordinates from the axes and returns a point with respect to the scene.

Parameters

coords
 The coordinates. Each coord is passed as a separate argument: ``ax.coords_to_point(1, 2, 3)``.

 Also accepts a list of coordinates

    ````ax.coords_to_point( [x_0, x_1, ...], [y_0, y_1, ...], ... )````

    ````ax.coords_to_point( [[x_0, y_0, z_0], [x_1, y_1, z_1]] )````

Returns

np.ndarray
 A point with respect to the scene's coordinate system.
 The shape of the array will be similar to the shape of the input.

Examples

.. code-block:: pycon

 >>> from manim import Axes
 >>> import numpy as np
 >>> ax = Axes()
 >>> np.around(ax.coords_to_point(1, 0, 0), 2)
 array([0.86, 0. , 0.])
 >>> np.around(ax.coords_to_point([[0, 1], [1, 1], [1, 0]]), 2)
 array([[0. , 0.75, 0.],
 [0.86, 0.75, 0.],
 [0.86, 0. , 0.]])
 >>> np.around(
 ... ax.coords_to_point([0, 1, 1], [1, 1, 0]), 2
 ...) # Transposed version of the above
 array([[0. , 0.86, 0.86],
 [0.75, 0.75, 0.],
 [0. , 0. , 0.]])

.. manim:: CoordsToPointExample
:save_last_frame:

class CoordsToPointExample(Scene):
 def construct(self):
 ax = Axes().add_coordinates()

 # a dot with respect to the axes
 dot_axes = Dot(ax.coords_to_point(2, 2), color=GREEN)
 lines = ax.get_lines_to_point(ax.c2p(2,2))

 # a dot with respect to the scene
 # the default plane corresponds to the coordinates of the scene.
 plane = NumberPlane()
 dot_scene = Dot((2,2,0), color=RED)

```

```

 self.add(plane, dot_scene, ax, dot_axes, lines)
"""
origin = self.x_axis.number_to_point(
 self._origin_shift([self.x_axis.x_min, self.x_axis.x_max]),
)
coords = np.asarray(coords)

if called like coords_to_point(1, 2, 3), then coords is a 1x3 array
transposed = False
if coords.ndim == 1:
 # original implementation of coords_to_point for performance in the legacy case
 result = np.array(origin)
 for axis, number in zip(self.get_axes(), coords):
 result += axis.number_to_point(number) - origin
 return result
if called like coords_to_point([1, 2, 3], [4, 5, 6]), then it shall be used as [1,4], [2,5], [3,6] and return the points as ([x_0,x_1],[y_0,y_1],[z_0,z_1])
elif coords.ndim == 2:
 coords = coords.T
 transposed = True
if called like coords_to_point(np.array([[1, 2, 3],[4,5,6]])), reduce dimension by 1
elif coords.ndim == 3:
 coords = np.squeeze(coords)
else the coords is a Nx1, Nx2, Nx3 array so we do not need to modify the array

points = origin + np.sum(
 [
 axis.number_to_point(number) - origin
 for number, axis in zip(coords.T, self.get_axes())
],
 axis=0,
)
if called with single coord, then return a point instead of a list of points
if transposed:
 return points.T
return points

def point_to_coords(self, point: Sequence[float]) -> np.ndarray:
 """Accepts a point from the scene and returns its coordinates with respect to the axes.

 Parameters

 point
 The point, i.e. ``RIGHT`` or ``[0, 1, 0]``.
 Also accepts a list of points as ``[RIGHT, [0, 1, 0]]``.

 Returns

 np.ndarray[float]
 The coordinates on the axes, i.e. ``[4.0, 7.0]``.
 Or a list of coordinates if `point` is a list of points.

 Examples

 .. code-block:: pycon

 >>> from manim import Axes, RIGHT
 >>> import numpy as np
 >>> ax = Axes(x_range=[0, 10, 2])
 >>> np.around(ax.point_to_coords(RIGHT), 2)
 array([5.83, 0.])
 >>> np.around(ax.point_to_coords([[0, 0, 1], [1, 0, 0]]), 2)
 array([[5., 0.],
 [5.83, 0.]])

 .. manim:: PointToCoordsExample
 :save_last_frame:

 class PointToCoordsExample(Scene):
 def construct(self):
 ax = Axes(x_range=[0, 10, 2]).add_coordinates()
 circ = Circle(radius=0.5).shift(UR * 2)

 # get the coordinates of the circle with respect to the axes
 coords = np.around(ax.point_to_coords(circ.get_right()), decimals=2)

 label = (
 Matrix([[coords[0]], [coords[1]]]).scale(0.75).next_to(circ, RIGHT)
)

 self.add(ax, circ, label, Dot(circ.get_right()))

 """
 point = np.asarray(point)
 result = np.asarray([axis.point_to_number(point) for axis in self.get_axes()])
 if point.ndim == 2:
 return result.T
 return result

def get_axes(self) -> VGroup:
 """Gets the axes.

 Returns

 :class:`~.VGroup`
 A pair of axes.
 """
 return self.axes

def get_axis_labels(
 self,
 x_label: float | str | Mobject = "x",
 y_label: float | str | Mobject = "y",
) -> VGroup:
 """Defines labels for the x-axis and y-axis of the graph.

 For increased control over the position of the labels,
 use :meth:`~.CoordinateSystem.get_x_axis_label` and
 :meth:`~.CoordinateSystem.get_y_axis_label`.

 Parameters

 x_label
 The label for the x_axis. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
 y_label
 The label for the y_axis. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.

 Returns

 :class:`~.VGroup`
 A :class:`~.VGroup` of the labels for the x_axis and y_axis.
 """

```

```

.. seealso::
 :meth:`~.CoordinateSystem.get_x_axis_label`
 :meth:`~.CoordinateSystem.get_y_axis_label`

Examples

.. manim:: GetAxisLabelsExample
 :save_last_frame:

 class GetAxisLabelsExample(Scene):
 def construct(self):
 ax = Axes()
 labels = ax.get_axis_labels(
 Tex("x-axis").scale(0.7), Text("y-axis").scale(0.45)
)
 self.add(ax, labels)

 self.axis_labels = VGroup(
 self.get_x_axis_label(x_label),
 self.get_y_axis_label(y_label),
)
 return self.axis_labels

def plot_line_graph(
 self,
 x_values: Iterable[float],
 y_values: Iterable[float],
 z_values: Iterable[float] | None = None,
 line_color: Color = YELLOW,
 add_vertex_dots: bool = True,
 vertex_dot_radius: float = DEFAULT_DOT_RADIUS,
 vertex_dot_style: dict | None = None,
 **kwargs,
) -> VDict:
 """Draws a line graph.

 The graph connects the vertices formed from zipping
 ``x_values``, ``y_values`` and ``z_values``. Also adds :class:`Dots <.Dot>` at the
 vertices if ``add_vertex_dots`` is set to ``True``.

 Parameters

 x_values
 Iterable of values along the x-axis.
 y_values
 Iterable of values along the y-axis.
 z_values
 Iterable of values (zeros if z_values is None) along the z-axis.
 line_color
 Color for the line graph.
 add_vertex_dots
 Whether or not to add :class:`~.Dot` at each vertex.
 vertex_dot_radius
 Radius for the :class:`~.Dot` at each vertex.
 vertex_dot_style
 Style arguments to be passed into :class:`~.Dot` at each vertex.
 kwargs
 Additional arguments to be passed into :class:`~.VMOobject`.

 Returns

 :class:`~.VDict`
 A VDict containing both the line and dots (if specified). The line can be accessed with: ``line_graph["line_graph"]``.
 The dots can be accessed with: ``line_graph["vertex_dots"]``.

Examples

.. manim:: LineGraphExample
 :save_last_frame:

 class LineGraphExample(Scene):
 def construct(self):
 plane = NumberPlane(
 x_range = (0, 7),
 y_range = (0, 5),
 x_length = 7,
 axis_config={"include_numbers": True},
)
 plane.center()
 line_graph = plane.plot_line_graph(
 x_values = [0, 1.5, 2, 2.8, 4, 6.25],
 y_values = [1, 3, 2.25, 4, 2.5, 1.75],
 line_color=GOLD_E,
 vertex_dot_style=dict(stroke_width=3, fill_color=PURPLE),
 stroke_width = 4,
)
 self.add(plane, line_graph)

 x_values, y_values = map(np.array, (x_values, y_values))
 if z_values is None:
 z_values = np.zeros(x_values.shape)

 line_graph = VDict()
 graph = VGroup(color=line_color, **kwargs)

 vertices = [
 self.coords_to_point(x, y, z)
 for x, y, z in zip(x_values, y_values, z_values)
]
 graph.set_points_as_corners(vertices)
 line_graph["line_graph"] = graph

 if add_vertex_dots:
 vertex_dot_style = vertex_dot_style or {}
 vertex_dots = VGroup(
 *(Dot(point=vertex, radius=vertex_dot_radius, **vertex_dot_style)
 for vertex in vertices)
)
 line_graph["vertex_dots"] = vertex_dots

 return line_graph

@staticmethod
def _origin_shift(axis_range: Sequence[float]) -> float:
 """Determines how to shift graph mobjects to compensate when 0 is not on the axis.

Parameters

axis_range
 The range of the axis : ``[x_min, x_max, x_step]``.
"""
 if axis_range[0] > 0:

```

```

 # min greater than 0
 return axis_range[0]
 if axis_range[1] < 0:
 # max less than 0
 return axis_range[1]
 else:
 return 0

class ThreeDaxes(Axes):
 """A 3-dimensional set of axes.

 Parameters

 x_range
 The ``[x_min, x_max, x_step]`` values of the x-axis.
 y_range
 The ``[y_min, y_max, y_step]`` values of the y-axis.
 z_range
 The ``[z_min, z_max, z_step]`` values of the z-axis.
 x_length
 The length of the x-axis.
 y_length
 The length of the y-axis.
 z_length
 The length of the z-axis.
 z_axis_config
 Arguments to be passed to :class:`~.NumberLine` that influence the z-axis.
 z_normal
 The direction of the normal.
 num_axis_pieces
 The number of pieces used to construct the axes.
 light_source
 The direction of the light source.
 depth
 Currently non-functional.
 gloss
 Currently non-functional.
 kwargs
 Additional arguments to be passed to :class:`Axes`.
 """

 def __init__(self,
 x_range: Sequence[float] | None = (-6, 6, 1),
 y_range: Sequence[float] | None = (-5, 5, 1),
 z_range: Sequence[float] | None = (-4, 4, 1),
 x_length: float | None = config.frame_height + 2.5,
 y_length: float | None = config.frame_height + 2.5,
 z_length: float | None = config.frame_height - 1.5,
 z_axis_config: dict | None = None,
 z_normal: Sequence[float] = DOWN,
 num_axis_pieces: int = 20,
 light_source: Sequence[float] = 9 * DOWN + 7 * LEFT + 10 * OUT,
 # opengl stuff (?)
 depth=None,
 gloss=0.5,
 **kwargs,
):
 super().__init__(
 x_range=x_range,
 x_length=x_length,
 y_range=y_range,
 y_length=y_length,
 **kwargs,
)

 self.z_range = z_range
 self.z_length = z_length

 self.z_axis_config = {}
 self._update_default_configs((self.z_axis_config,), (z_axis_config,))
 self.z_axis_config = merge_dicts_recursively(
 self.axis_config,
 self.z_axis_config,
)

 self.z_normal = z_normal
 self.num_axis_pieces = num_axis_pieces

 self.light_source = light_source
 self.dimension = 3

 if self.z_axis_config.get("scaling") is None or isinstance(
 self.z_axis_config.get("scaling"), LinearBase
):
 self.z_axis_config["exclude_origin_tick"] = True
 else:
 self.z_axis_config["exclude_origin_tick"] = False

 z_axis = self._create_axis(self.z_range, self.z_axis_config, self.z_length)

 # [ax.x_min, ax.x_max] used to account for LogBase() scaling
 # where ax.x_range[0] != ax.x_min
 z_origin = self._origin_shift([z_axis.x_min, z_axis.x_max])

 z_axis.rotate_about_number(z_origin, -PI / 2, UP)
 z_axis.rotate_about_number(z_origin, angle_of_vector(self.z_normal))
 z_axis.shift(-z_axis.number_to_point(z_origin))
 z_axis.shift(
 self.x_axis.number_to_point(
 self._origin_shift([self.x_axis.x_min, self.x_axis.x_max]),
),
)

 self.axes.add(z_axis)
 self.add(z_axis)
 self.z_axis = z_axis

 if config.renderer == RendererType.CAIRO:
 self._add_3d_pieces()
 self._set_axis_shading()

 def _add_3d_pieces(self):
 for axis in self.axes:
 axis.pieces = VGroup(*axis.get_pieces(self.num_axis_pieces))
 axis.add(axis.pieces)
 axis.set_stroke(width=0, family=False)
 axis.set_shade_in_3d(True)

 def _set_axis_shading(self):
 def make_func(axis):
 vect = self.light_source
 return lambda: (

```

```

 axis.get_edge_center(-vect),
 axis.get_edge_center(vect),
)

 for axis in self:
 for submob in axis.family_members_with_points():
 submob.get_gradient_start_and_end_points = make_func(axis)
 submob.get_unit_normal = lambda a: np.ones(3)
 submob.set_sheen(0.2)

def get_y_axis_label(
 self,
 label: float | str | Mobject,
 edge: Sequence[float] = UR,
 direction: Sequence[float] = UR,
 buff: float = SMALL_BUFF,
 rotation=PI / 2,
 rotation_axis=OUT,
 **kwargs,
) -> Mobject:
 """Generate a y-axis label.

Parameters

label
 The label. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
edge
 The edge of the y-axis to which the label will be added, by default ``UR``.
direction
 Allows for further positioning of the label from an edge, by default ``UR``.
buff
 The distance of the label from the line, by default ``SMALL_BUFF``.
rotation
 The angle at which to rotate the label, by default ``PI/2``.
rotation_axis
 The axis about which to rotate the label, by default ``OUT``.

Returns

:class:`~.Mobject`
 The positioned label.

Examples

.. manim:: GetYAxisLabelExample
 :save_last_frame:

 class GetYAxisLabelExample(ThreeDScene):
 def construct(self):
 ax = ThreeDAxes()
 lab = ax.get_y_axis_label(Tex("y-label"))
 self.set_camera_orientation(phi=2*PI/5, theta=PI/5)
 self.add(ax, lab)

 positioned_label = self._get_axis_label(
 label, self.get_y_axis(), edge, direction, buff=buff, **kwargs
)
 positioned_label.rotate(rotation, axis=rotation_axis)
 return positioned_label

def get_z_axis_label(
 self,
 label: float | str | Mobject,
 edge: Sequence[float] = OUT,
 direction: Sequence[float] = RIGHT,
 buff: float = SMALL_BUFF,
 rotation=PI / 2,
 rotation_axis=RIGHT,
 **kwargs,
) -> Mobject:
 """Generate a z-axis label.

Parameters

label
 The label. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
edge
 The edge of the z-axis to which the label will be added, by default ``OUT``.
direction
 Allows for further positioning of the label from an edge, by default ``RIGHT``.
buff
 The distance of the label from the line, by default ``SMALL_BUFF``.
rotation
 The angle at which to rotate the label, by default ``PI/2``.
rotation_axis
 The axis about which to rotate the label, by default ``RIGHT``.

Returns

:class:`~.Mobject`
 The positioned label.

Examples

.. manim:: GetZAxisLabelExample
 :save_last_frame:

 class GetZAxisLabelExample(ThreeDScene):
 def construct(self):
 ax = ThreeDAxes()
 lab = ax.get_z_axis_label(Tex("z-label"))
 self.set_camera_orientation(phi=2*PI/5, theta=PI/5)
 self.add(ax, lab)

 positioned_label = self._get_axis_label(
 label, self.get_z_axis(), edge, direction, buff=buff, **kwargs
)
 positioned_label.rotate(rotation, axis=rotation_axis)
 return positioned_label

def get_axis_labels(
 self,
 x_label: float | str | Mobject = "x",
 y_label: float | str | Mobject = "y",
 z_label: float | str | Mobject = "z",
) -> VGroup:
 """Defines labels for the x_axis and y_axis of the graph.

For increased control over the position of the labels,
use :meth:`~.CoordinateSystem.get_x_axis_label`,
:meth:`~.ThreeDAxes.get_y_axis_label`, and
:meth:`~.ThreeDAxes.get_z_axis_label`.

```

```

Parameters

x_label
 The label for the x_axis. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
y_label
 The label for the y_axis. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.
z_label
 The label for the z_axis. Defaults to :class:`~.MathTex` for ``str`` and ``float`` inputs.

Returns

:class:`~.VGroup`
 A :class:`~.VGroup` of the labels for the x_axis, y_axis, and z_axis.

..seealso::
 :meth:`~.CoordinateSystem.get_x_axis_label`
 :meth:`~.ThreeDAxes.get_y_axis_label`
 :meth:`~.ThreeDAxes.get_z_axis_label`

Examples

.. manim:: GetAxisLabelsExample
 :save_last_frame:

 class GetAxisLabelsExample(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=2*PI/5, theta=PI/5)
 axes = ThreeDAxes()
 labels = axes.get_axis_labels(
 Tex("x-axis").scale(0.7), Text("y-axis").scale(0.45), Text("z-axis").scale(0.45)
)
 self.add(axes, labels)

 self.axis_labels = VGroup(
 self.get_x_axis_label(x_label),
 self.get_y_axis_label(y_label),
 self.get_z_axis_label(z_label),
)
 return self.axis_labels

class NumberPlane(Axes):
 """Creates a cartesian plane with background lines.

Parameters

x_range
 The ``[x_min, x_max, x_step]`` values of the plane in the horizontal direction.
y_range
 The ``[y_min, y_max, y_step]`` values of the plane in the vertical direction.
x_length
 The width of the plane.
y_length
 The height of the plane.
background_line_style
 Arguments that influence the construction of the background lines of the plane.
faded_line_style
 Similar to :attr:`background_line_style`, affects the construction of the scene's background lines.
faded_line_ratio
 Determines the number of boxes within the background lines: :code:`2` = 4 boxes, :code:`3` = 9 boxes.
make_smooth_after_applying_functions
 Currently non-functional.
kwargs
 Additional arguments to be passed to :class:`Axes`.

.. note::
 If :attr:`x_length` or :attr:`y_length` are not defined, they are automatically calculated such that
 one unit on each axis is one Manim unit long.

Examples

.. manim:: NumberPlaneExample
 :save_last_frame:

 class NumberPlaneExample(Scene):
 def construct(self):
 number_plane = NumberPlane(
 background_line_styles={
 "stroke_color": TEAL,
 "stroke_width": 4,
 "stroke_opacity": 0.6
 }
)
 self.add(number_plane)

 .. manim:: NumberPlaneScaled
 :save_last_frame:

 class NumberPlaneScaled(Scene):
 def construct(self):
 number_plane = NumberPlane(
 x_range=(-4, 11, 1),
 y_range=(-3, 3, 1),
 x_length=5,
 y_length=2,
).move_to(LEFT*3)

 number_plane_scaled_y = NumberPlane(
 x_range=(-4, 11, 1),
 x_length=5,
 y_length=4,
).move_to(RIGHT*3)

 self.add(number_plane)
 self.add(number_plane_scaled_y)

 def __init__(self, x_range: Sequence[float], y_range: Sequence[float], x_length: float | None = None, y_length: float | None = None, **kwargs):
 super().__init__(**kwargs)
 if x_length is None:
 x_length = config["frame_x_radius"] * 2
 if y_length is None:
 y_length = config["frame_y_radius"] * 2

```

```

y_length: float | None = None,
background_line_style: dict | None = None,
faded_line_style: dict | None = None,
faded_line_ratio: int = 1,
make_smooth_after_applying_functions: bool = True,
**kwargs,
):
 # configs
 self.axis_config = {
 "stroke_width": 2,
 "include_ticks": False,
 "include_tip": False,
 "line_to_number_buff": SMALL_BUFF,
 "label_direction": DR,
 "font_size": 24,
 }
 self.y_axis_config = {"label_direction": DR}
 self.background_line_style = {
 "stroke_color": BLUE_D,
 "stroke_width": 2,
 "stroke_opacity": 1,
 }
 self._update_default_configs(
 (self.axis_config, self.y_axis_config, self.background_line_style),
 (
 kwargs.pop("axis_config", None),
 kwargs.pop("y_axis_config", None),
 background_line_style,
),
)
 # Defaults to a faded version of line_config
 self.faded_line_style = faded_line_style
 self.faded_line_ratio = faded_line_ratio
 self.make_smooth_after_applying_functions = make_smooth_after_applying_functions

 # init
 super().__init__(
 x_range=x_range,
 y_range=y_range,
 x_length=x_length,
 y_length=y_length,
 axis_config=self.axis_config,
 y_axis_config=self.y_axis_config,
 **kwargs,
)

 self._init_background_lines()

def _init_background_lines(self):
 """Will init all the lines of NumberPlanes (faded or not)"""
 if self.faded_line_style is None:
 style = dict(self.background_line_style)
 # For anything numerical, like stroke_width
 # and stroke_opacity, chop it in half
 for key in style:
 if isinstance(style[key], numbers.Number):
 style[key] *= 0.5
 self.faded_line_style = style

 self.background_lines, self.faded_lines = self._get_lines()

 self.background_lines.set_style(
 **self.background_line_style,
)
 self.faded_lines.set_style(
 **self.faded_line_style,
)
 self.add_to_back(
 self.faded_lines,
 self.background_lines,
)

def _get_lines(self) -> tuple[VGroup, VGroup]:
 """Generate all the lines, faded and not faded.
 Two sets of lines are generated: one parallel to the X-axis, and parallel to the Y-axis.

 Returns

 Tuple[:class:`~.VGroup`, :class:`~.VGroup`]
 The first (i.e the non faded lines) and second (i.e the faded lines) sets of lines, respectively.
 """
 x_axis = self.get_x_axis()
 y_axis = self.get_y_axis()

 x_lines1, x_lines2 = self._get_lines_parallel_to_axis(
 x_axis,
 y_axis,
 self.y_axis.x_range[2],
 self.faded_line_ratio,
)

 y_lines1, y_lines2 = self._get_lines_parallel_to_axis(
 y_axis,
 x_axis,
 self.x_axis.x_range[2],
 self.faded_line_ratio,
)

 # TODO this was added so that we can run tests on NumberPlane
 # In the future these attributes will be tacked onto self.background_lines
 self.x_lines = x_lines1
 self.y_lines = y_lines1
 lines1 = VGroup(*x_lines1, *y_lines1)
 lines2 = VGroup(*x_lines2, *y_lines2)

 return lines1, lines2

def _get_lines_parallel_to_axis(
 self,
 axis_parallel_to: NumberLine,
 axis_perpendicular_to: NumberLine,
 freq: float,
 ratio_faded_lines: int,
) -> tuple[VGroup, VGroup]:
 """Generate a set of lines parallel to an axis.

 Parameters

 axis_parallel_to
 The axis with which the lines will be parallel.
 axis_perpendicular_to
 The axis with which the lines will be perpendicular.
 ratio_faded_lines
 The ratio of faded lines to non-faded lines.
 """

```

```

The ratio between the space between faded lines and the space between non-faded lines.
freq
 Frequency of non-faded lines (number of non-faded lines per graph unit).

Returns

Tuple[:class:`~.VGroup`, :class:`~.VGroup`]
 The first (i.e. the non-faded lines parallel to `axis_parallel_to`) and second
 (i.e. the faded lines parallel to `axis_parallel_to`) sets of lines, respectively.
"""

line = Line(axis_parallel_to.get_start(), axis_parallel_to.get_end())
if ratio_faded_lines == 0: # don't show faded lines
 ratio_faded_lines = 1 # i.e. set ratio to 1
step = (1 / ratio_faded_lines) * freq
lines1 = VGroup()
lines2 = VGroup()
unit_vector_axis_perp_to = axis_perpendicular_to.get_unit_vector()

need to unpack all three values
x_min, x_max, _ = axis_perpendicular_to.x_range

account for different axis scalings (logarithmic), where
negative values do not exist and [-2, 4] should output lines
similar to [0, 6]
if axis_perpendicular_to.x_min > 0 and x_min < 0:
 x_min, x_max = (0, np.abs(x_min) + np.abs(x_max))

min/max used in case range does not include 0. i.e. if (2,6):
the range becomes (0,4), not (0,6).
ranges = (
 [0],
 np.arange(step, min(x_max - x_min, x_max), step),
 np.arange(-step, max(x_min - x_max, x_min), -step),
)

for inputs in ranges:
 for k, x in enumerate(inputs):
 new_line = line.copy()
 new_line.shift(unit_vector_axis_perp_to * x)
 if (k + 1) % ratio_faded_lines == 0:
 lines1.add(new_line)
 else:
 lines2.add(new_line)
return lines1, lines2

def get_vector(self, coords: Sequence[float], **kwargs):
 kwargs["buff"] = 0
 return Arrow(
 self.coords_to_point(0, 0), self.coords_to_point(*coords), **kwargs
)

def prepare_for_nonlinear_transform(self, num_inserted_curves: int = 50):
 for mob in self.family_members_with_points():
 num_curves = mob.get_num_curves()
 if num_inserted_curves > num_curves:
 mob.insert_n_curves(num_inserted_curves - num_curves)
 return self

class PolarPlane(Axes):
 """Creates a polar plane with background lines.

 Parameters

 azimuth_step
 The number of divisions in the azimuth (also known as the `angular coordinate` or `polar angle`). If ``None`` is specified then it will use the default specified by ``azimuth_units``:
 - ``"PI radians"`` or ``"TAU radians"``: 20
 - ``"degrees"``: 36
 - ``"radians"``: 40
 - ``"None"``: 1

 A non-integer value will result in a partial division at the end of the circle.

 size
 The diameter of the plane.

 radius_step
 The distance between faded radius lines.

 radius_max
 The maximum value of the radius.

 azimuth_units
 Specifies a default labelling system for the azimuth. Choices are:
 - ``"PI radians"``: Fractional labels in the interval :math:`\left[0, 2\pi\right)` with :math:`\pi` as a constant.
 - ``"TAU radians"``: Fractional labels in the interval :math:`\left[0, \tau\right)` (where :math:`\tau = 2\pi` with :math:`\tau` as a constant.
 - ``"degrees"``: Decimal labels in the interval :math:`\left[0, 360^\circ\right)` with a degree (:math:`^\circ`(\circ)) symbol.
 - ``"radians"``: Decimal labels in the interval :math:`\left[0, 400\right)` with a superscript "g" (:math:`^g`(\{g\})).
 - ``"None"``: Decimal labels in the interval :math:`\left[0, 1\right)`.

 azimuth_compact_fraction
 If the ``azimuth_units`` choice has fractional labels, choose whether to combine the constant in a compact form :math:`\tfrac{x}{y}` as opposed to :math:`\frac{x}{y}\pi` where :math:`\pi` is the constant.

 azimuth_offset
 The angle offset of the azimuth, expressed in radians.

 azimuth_direction
 The direction of the azimuth.
 - ``"CW"``: Clockwise.
 - ``"CCW"``: Anti-clockwise.

 azimuth_label_buff
 The buffer for the azimuth labels.

 azimuth_label_font_size
 The font size of the azimuth labels.

 radius_config
 The axis config for the radius.

 Examples

 .. manim:: PolarPlaneExample
 :ref_classes: PolarPlane
 :save_last_frame:

 class PolarPlaneExample(Scene):
 def construct(self):

```

```

polarplane_pi = PolarPlane(
 azimuth_units="PI radians",
 size=6,
 azimuth_label_font_size=33.6,
 radius_config={"font_size": 33.6},
).add_coordinates()
self.add(polarplane_pi)
"""

def __init__(self,
radius_max: float = config["frame_y_radius"],
size: float | None = None,
radius_step: float = 1,
azimuth_step: float | None = None,
azimuth_units: str | None = "PI radians",
azimuth_compact_fraction: bool = True,
azimuth_offset: float = 0,
azimuth_direction: str = "CCW",
azimuth_label_buff: float = SMALL_BUFF,
azimuth_label_font_size: float = 24,
radius_config: dict | None = None,
background_line_style: dict | None = None,
faded_line_style: dict | None = None,
faded_line_ratio: int = 1,
make_smooth_after_applying_functions: bool = True,
**kwargs,
):
 # error catching
 if azimuth_units in ["PI radians", "TAU radians", "degrees", "gradians", None]:
 self.azimuth_units = azimuth_units
 else:
 raise ValueError(
 "Invalid azimuth units. Expected one of: PI radians, TAU radians, degrees, gradians or None.",
)

 if azimuth_direction in ["CW", "CCW"]:
 self.azimuth_direction = azimuth_direction
 else:
 raise ValueError("Invalid azimuth units. Expected one of: CW, CCW.")

 # configs
 self.radius_config = {
 "stroke_width": 2,
 "include_ticks": False,
 "include_tip": False,
 "line_to_number_buff": SMALL_BUFF,
 "label_direction": DL,
 "font_size": 24,
 }

 self.background_line_style = {
 "stroke_color": BLUE_D,
 "stroke_width": 2,
 "stroke_opacity": 1,
 }

 self.azimuth_step = (
 {
 {
 "PI radians": 20,
 "TAU radians": 20,
 "degrees": 36,
 "gradians": 40,
 None: 1,
 }[azimuth_units]
 }
 if azimuth_step is None
 else azimuth_step
)

 self._update_default_configs(
 (self.radius_config, self.background_line_style),
 (radius_config, background_line_style),
)

 # Defaults to a faded version of line_config
 self.faded_line_style = faded_line_style
 self.faded_line_ratio = faded_line_ratio
 self.make_smooth_after_applying_functions = make_smooth_after_applying_functions
 self.azimuth_offset = azimuth_offset
 self.azimuth_label_buff = azimuth_label_buff
 self.azimuth_label_font_size = azimuth_label_font_size
 self.azimuth_compact_fraction = azimuth_compact_fraction

 # init
 super().__init__(
 x_range=np.array((-radius_max, radius_max, radius_step)),
 y_range=np.array((-radius_max, radius_max, radius_step)),
 x_length=size,
 y_length=size,
 axis_config=self.radius_config,
 **kwargs,
)

 self._init_background_lines()

def _init_background_lines(self):
 """Will init all the lines of NumberPlanes (faded or not)"""
 if self.faded_line_style is None:
 style = dict(self.background_line_style)
 # For anything numerical, like stroke_width
 # and stroke_opacity, chop it in half
 for key in style:
 if isinstance(style[key], numbers.Number):
 style[key] *= 0.5
 self.faded_line_style = style

 self.background_lines, self.faded_lines = self._get_lines()
 self.background_lines.set_style(
 **self.background_line_style,
)
 self.faded_lines.set_style(
 **self.faded_line_style,
)
 self.add_to_back(
 self.faded_lines,
 self.background_lines,
)

def _get_lines(self) -> tuple[VGroup, VGroup]:
 """Generate all the lines and circles, faded and not faded.

 Returns

```

```

Tuple[:class: `~.VGroup`, :class: `~.VGroup`]
 The first (i.e the non faded lines and circles) and second (i.e the faded lines and circles) sets of lines and circles, respectively.
"""
center = self.get_origin()
ratio_faded_lines = self.faded_line_ratio
offset = self.azimuth_offset

if ratio_faded_lines == 0: # don't show faded lines
 ratio_faded_lines = 1 # i.e. set ratio to 1
rstep = (1 / ratio_faded_lines) * self.x_axis.x_range[2]
astep = (1 / ratio_faded_lines) * (TAU * (1 / self.azimuth_step))
rlines1 = VGroup()
rlines2 = VGroup()
alines1 = VGroup()
alines2 = VGroup()

rinput = np.arange(0, self.x_axis.x_range[1] + rstep, rstep)
ainput = np.arange(0, TAU, astep)

unit_vector = self.x_axis.get_unit_vector()[0]

for k, x in enumerate(rinput):
 new_line = Circle(radius=x * unit_vector)
 if k % ratio_faded_lines == 0:
 alines1.add(new_line)
 else:
 alines2.add(new_line)

line = Line(center, self.get_x_axis().get_end())

for k, x in enumerate(ainput):
 new_line = line.copy()
 new_line.rotate(x + offset, about_point=center)
 if k % ratio_faded_lines == 0:
 rlines1.add(new_line)
 else:
 rlines2.add(new_line)

lines1 = VGroup(*rlines1, *alines1)
lines2 = VGroup(*rlines2, *alines2)
return lines1, lines2

def get_axes(self) -> VGroup:
 """Gets the axes.

 Returns

 :class: `~.VGroup`
 A pair of axes.
 """
 return self.axes

def get_vector(self, coords, **kwargs):
 kwargs["buff"] = 0
 return Arrow(
 self.coords_to_point(0, 0), self.coords_to_point(*coords), **kwargs
)

def prepare_for_nonlinear_transform(self, num_inserted_curves=50):
 for mob in self.family_members_with_points():
 num_curves = mob.get_num_curves()
 if num_inserted_curves > num_curves:
 mob.insert_n_curves(num_inserted_curves - num_curves)
 return self

def get_coordinate_labels(
 self,
 r_values: Iterable[float] | None = None,
 a_values: Iterable[float] | None = None,
 **kwargs,
) -> VDict:
 """Gets labels for the coordinates

 Parameters

 r_values
 Iterable of values along the radius, by default None.
 a_values
 Iterable of values along the azimuth, by default None.

 Returns

 VDict
 Labels for the radius and azimuth values.
 """
 if r_values is None:
 r_values = [r for r in self.get_x_axis().get_tick_range() if r >= 0]
 if a_values is None:
 a_values = np.arange(0, 1, 1 / self.azimuth_step)
 r_mobs = self.get_x_axis().add_numbers(r_values)
 if self.azimuth_direction == "CCW":
 d = 1
 elif self.azimuth_direction == "CW":
 d = -1
 else:
 raise ValueError("Invalid azimuth direction. Expected one of: CW, CCW")
 a_points = [
 {
 "label": i,
 "point": np.array([
 self.get_right()[0]
 * np.cos(d * (i * TAU) + self.azimuth_offset),
 self.get_right()[0]
 * np.sin(d * (i * TAU) + self.azimuth_offset),
 0,
]),
 },
 for i in a_values
]
 if self.azimuth_units == "PI radians" or self.azimuth_units == "TAU radians":
 a_tex = [
 self.get_radian_label(
 i["label"],
 font_size=self.azimuth_label_font_size,
).next_to(
 i["point"],
 direction=i["point"],
 aligned_edge=i["point"],
 buff=self.azimuth_label_buff,
)
 for i in a_points
]

```

```

 elif self.azimuth_units == "degrees":
 a_tex = [
 MathTex(
 f'{360 * i["label"]}:g)' + r"^\circ",
 font_size=self.azimuth_label_font_size,
).next_to(
 i["point"],
 direction=i["point"],
 aligned_edge=i["point"],
 buff=self.azimuth_label_buff,
)
 for i in a_points
]
 elif self.azimuth_units == "radians":
 a_tex = [
 MathTex(
 f'{400 * i["label"]}:g)' + r"^\text{g}",
 font_size=self.azimuth_label_font_size,
).next_to(
 i["point"],
 direction=i["point"],
 aligned_edge=i["point"],
 buff=self.azimuth_label_buff,
)
 for i in a_points
]
 elif self.azimuth_units is None:
 a_tex = [
 MathTex(
 f'{i["label"]}:g',
 font_size=self.azimuth_label_font_size,
).next_to(
 i["point"],
 direction=[("point"),
 aligned_edge=i["point"],
 buff=self.azimuth_label_buff,
]
 for i in a_points
]
 a_mobs = VGroup(*a_tex)
 self.coordinate_labels = VGroup(r_mobs, a_mobs)
 return self.coordinate_labels

def add_coordinates(
 self,
 r_values: Iterable[float] | None = None,
 a_values: Iterable[float] | None = None,
):
 """Adds the coordinates.

 Parameters

 r_values
 Iterable of values along the radius, by default None.
 a_values
 Iterable of values along the azimuth, by default None.
 """
 self.add(self.get_coordinate_labels(r_values, a_values))
 return self

def get_radian_label(self, number, font_size=24, **kwargs):
 constant_label = {"PI radians": r"\pi", "TAU radians": r"\tau"}[
 self.azimuth_units
]
 division = number * {"PI radians": 2, "TAU radians": 1}[self.azimuth_units]
 frac = fr.Fraction(division).limit_denominator(max_denominator=100)
 if frac.numerator == 0 & frac.denominator == 0:
 string = r"0"
 elif frac.numerator == 1 and frac.denominator == 1:
 string = constant_label
 elif frac.numerator == 1:
 if self.azimuth_compact_fraction:
 string = (
 r"\tfrac{" + constant_label + r"}{" + str(frac.denominator) + "}"
)
 else:
 string = r"\tfrac{1}{" + str(frac.denominator) + "}" + constant_label
 elif frac.denominator == 1:
 string = str(frac.numerator) + constant_label
 else:
 if self.azimuth_compact_fraction:
 string = (
 r"\tfrac{" + str(frac.numerator)
 + constant_label
 + r"}{" + str(frac.denominator)
 + r"}"
)
 else:
 string = (
 r"\tfrac{" + str(frac.numerator)
 + r"}{" + str(frac.denominator)
 + r"}" + constant_label
)
 return MathTex(string, font_size=font_size, **kwargs)

class ComplexPlane(NumberPlane):
 """A :class:`~.NumberPlane` specialized for use with complex numbers.

 Examples

 .. manim:: ComplexPlaneExample
 :save_last_frame:
 :ref_classes: Dot MathTex

 class ComplexPlaneExample(Scene):
 def construct(self):
 plane = ComplexPlane().add_coordinates()
 self.add(plane)
 d1 = Dot(plane.n2p(2 + 1j), color=YELLOW)
 d2 = Dot(plane.n2p(-3 - 2j), color=YELLOW)
 label1 = MathTex("2+i").next_to(d1, UR, 0.1)
 label2 = MathTex("-3-2i").next_to(d2, UR, 0.1)
 self.add(
 d1,
 label1,
 d2,
 label2,

```

```

)
"""

def __init__(self, **kwargs):
 super().__init__(
 **kwargs,
)

def number_to_point(self, number: float | complex) -> np.ndarray:
 """Accepts a float/complex number and returns the equivalent point on the plane.

 Parameters

 number
 The number. Can be a float or a complex number.

 Returns

 np.ndarray
 The point on the plane.
 """

 number = complex(number)
 return self.coords_to_point(number.real, number.imag)

def n2p(self, number: float | complex) -> np.ndarray:
 """Abbreviation for :meth:`number_to_point`."""
 return self.number_to_point(number)

def point_to_number(self, point: Sequence[float]) -> complex:
 """Accepts a point and returns a complex number equivalent to that point on the plane.

 Parameters

 point
 The point in manim's coordinate-system

 Returns

 complex
 A complex number consisting of real and imaginary components.
 """

 x, y = self.point_to_coords(point)
 return complex(x, y)

def p2n(self, point: Sequence[float]) -> complex:
 """Abbreviation for :meth:`point_to_number`."""
 return self.point_to_number(point)

def _get_default_coordinate_values(self) -> list[float | complex]:
 """Generate a list containing the numerical values of the plane's labels.

 Returns

 List[float | complex]
 A list of floats representing the x-axis and complex numbers representing the y-axis.
 """

 x_numbers = self.get_x_axis().get_tick_range()
 y_numbers = self.get_y_axis().get_tick_range()
 y_numbers = [complex(0, y) for y in y_numbers if y != 0]
 return [*x_numbers, *y_numbers]

def get_coordinate_labels(
 self, *numbers: Iterable[float | complex], **kwargs
) -> VGroup:
 """Generates the :class:`~.DecimalNumber` mobjects for the coordinates of the plane.

 Parameters

 numbers
 An iterable of floats/complex numbers. Floats are positioned along the x-axis, complex numbers along the y-axis.
 kwargs
 Additional arguments to be passed to :meth:`~.NumberLine.get_number_mobject`, i.e. :class:`~.DecimalNumber`.

 Returns

 :class:`~.VGroup`
 A :class:`~.VGroup` containing the positioned label mobjects.
 """

 # TODO: Make this work the same as coord_sys.add_coordinates()
 if len(numbers) == 0:
 numbers = self._get_default_coordinate_values()

 self.coordinate_labels = VGroup()
 for number in numbers:
 z = complex(number)
 if abs(z.imag) > abs(z.real):
 axis = self.get_y_axis()
 value = z.imag
 kwargs["unit"] = "i"
 else:
 axis = self.get_x_axis()
 value = z.real
 number_mob = axis.get_number_mobject(value, **kwargs)
 self.coordinate_labels.add(number_mob)
 return self.coordinate_labels

def add_coordinates(self, *numbers: Iterable[float | complex], **kwargs):
 """Adds the labels produced from :meth:`~.NumberPlane.get_coordinate_labels` to the plane.

 Parameters

 numbers
 An iterable of floats/complex numbers. Floats are positioned along the x-axis, complex numbers along the y-axis.
 kwargs
 Additional arguments to be passed to :meth:`~.NumberLine.get_number_mobject`, i.e. :class:`~.DecimalNumber`.
 """

 self.add(self.get_coordinate_labels(*numbers, **kwargs))
 return self

```

# matrix.py

```
r"""Mobjects representing matrices.

Examples

.. manim:: MatrixExamples
 :save_last_frame:

 class MatrixExamples(Scene):
 def construct(self):
 m0 = Matrix([["\pi", 0], [-1, 1]])
 m1 = IntegerMatrix([[1.5, 0], [12, -1.3]],
 left_bracket="(",
 right_bracket=")")
 m2 = DecimalMatrix(
 [[3.456, 2.122], [33.2244, 12.33]],
 element_to_mobobject_config={"num_decimal_places": 2},
 left_bracket="\{",
 right_bracket="\}")
 m3 = MobjectMatrix(
 [[Circle().scale(0.3), Square().scale(0.3)],
 [MathTex("\pi").scale(2), Star().scale(0.3)]],
 left_bracket="\langle",
 right_bracket="\rangle")
 g = Group(m0, m1, m2, m3).arrange_in_grid(buff=2)
 self.add(g)

from __future__ import annotations

__all__ = [
 "Matrix",
 "DecimalMatrix",
 "IntegerMatrix",
 "MobjectMatrix",
 "matrix_to_tex_string",
 "matrix_to_mobobject",
 "get_det_text",
]
]

import itertools as it
from typing import Iterable, Sequence

import numpy as np

from manim.mobject.mobject import Mobject
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.mobject.text.numbers import DecimalNumber, Integer
from manim.mobject.text.tex_mobobject import MathTex, Tex

from ..constants import *
from ..mobject.types.vectorized_mobject import VGroup, VMobject

TO DO : The following two functions are not used in this file.
Not sure if we should keep it or not.

def matrix_to_tex_string(matrix):
 matrix = np.array(matrix).astype("str")
 if matrix.ndim == 1:
 matrix = matrix.reshape((matrix.size, 1))
 n_rows, n_cols = matrix.shape
 prefix = "\left[\begin{array}{%s}" % ("c" * n_cols)
 suffix = "\end{array} \right]"
 rows = [" ".join(row) for row in matrix]
 return prefix + " \\ ".join(rows) + suffix

def matrix_to_mobobject(matrix):
 return MathTex(matrix_to_tex_string(matrix))

class Matrix(VMobject, metaclass=ConvertToOpenGL):
 """A mobject that displays a matrix on the screen.

 Parameters

 matrix
 A numpy 2d array or list of lists.
 v_buff
 """

 def __init__(self, matrix, v_buff=1, **kwargs):
 super().__init__(**kwargs)
 self.matrix = matrix
 self.v_buff = v_buff
 self.set_color_by_luminance()
 self.set_stroke(width=2)
```

```

 Vertical distance between elements, by default 0.8.
h_buff
 Horizontal distance between elements, by default 1.3.
bracket_h_buff
 Distance of the brackets from the matrix, by default ``MED_SMALL_BUFF``.
bracket_v_buff
 Height of the brackets, by default ``MED_SMALL_BUFF``.
add_background_rectangles_to_entries
 ``True`` if should add background rectangles to entries, by default ``False``.
include_background_rectangle
 ``True`` if should include background rectangle, by default ``False``.
element_to_mobject
 The mobject class used to construct the elements, by default :class:`~.MathTex`.
element_to_mobject_config
 Additional arguments to be passed to the constructor in ``element_to_mobject``, by default ``{}``.
element_alignment_corner
 The corner to which elements are aligned, by default ``DR``.
left_bracket
 The left bracket type, by default `` "[" ``.
right_bracket
 The right bracket type, by default `` "]" ``.
stretch_brackets
 ``True`` if should stretch the brackets to fit the height of matrix contents, by default ``True``.
bracket_config
 Additional arguments to be passed to :class:`~.MathTex` when constructing the brackets.

```

## Examples

-----  
The first example shows a variety of uses of this module while the second example explains the use of the options `add\_background\_rectangles\_to\_entries` and `include\_background\_rectangle`.

```

.. manim:: MatrixExamples
:save_last_frame:

class MatrixExamples(Scene):
 def construct(self):
 m0 = Matrix([[2, "\pi"], [-1, 1]])
 m1 = Matrix([[2, 0, 4], [-1, 1, 5]],
 v_buff=1.3,
 h_buff=0.8,
 bracket_h_buff=SMALL_BUFF,
 bracket_v_buff=SMALL_BUFF,
 left_bracket="\\"(",
 right_bracket="\\")")
 m1.add(SurroundingRectangle(m1.get_columns()[1]))
 m2 = Matrix([[2, 1], [-1, 3]],
 element_alignment_corner=UL,
 left_bracket="(",
 right_bracket=")")
 m3 = Matrix([[2, 1], [-1, 3]],
 left_bracket="\\\langle",
 right_bracket="\\\rangle")
 m4 = Matrix([[2, 1], [-1, 3]],
).set_column_colors(RED, GREEN)
 m5 = Matrix([[2, 1], [-1, 3]],
).set_row_colors(RED, GREEN)
 g = Group(
 m0, m1, m2, m3, m4, m5
).arrange_in_grid(buff=2)
 self.add(g)

.. manim:: BackgroundRectanglesExample
:save_last_frame:

class BackgroundRectanglesExample(Scene):
 def construct(self):
 background= Rectangle().scale(3.2)
 background.set_fill(opacity=.5)
 background.set_color([TEAL, RED, YELLOW])
 self.add(background)
 m0 = Matrix([[12, -30], [-1, 15]],
 add_background_rectangles_to_entries=True)
 m1 = Matrix([[2, 0], [-1, 1]],
 include_background_rectangle=True)
 m2 = Matrix([[12, -30], [-1, 15]])
 g = Group(m0, m1, m2).arrange(buff=2)
 self.add(g)
"""

def __init__(self,
 matrix: Iterable,
 v_buff: float = 0.8,
 h_buff: float = 1.3,

```

```

bracket_h_buff: float = MED_SMALL_BUFF,
bracket_v_buff: float = MED_SMALL_BUFF,
add_background_rectangles_to_entries: bool = False,
include_background_rectangle: bool = False,
element_to_mob: type[MathTex] = MathTex,
element_to_mob_config: dict = {},
element_alignment_corner: Sequence[float] = DR,
left_bracket: str = "[",
right_bracket: str = "]",
stretch_brackets: bool = True,
bracket_config: dict = {},
**kwargs,
):
 self.v_buff = v_buff
 self.h_buff = h_buff
 self.bracket_h_buff = bracket_h_buff
 self.bracket_v_buff = bracket_v_buff
 self.add_background_rectangles_to_entries = add_background_rectangles_to_entries
 self.include_background_rectangle = include_background_rectangle
 self.element_to_mob = element_to_mob
 self.element_to_mob_config = element_to_mob_config
 self.element_alignment_corner = element_alignment_corner
 self.left_bracket = left_bracket
 self.right_bracket = right_bracket
 self.stretch_brackets = stretch_brackets
 super().__init__(**kwargs)
 mob_matrix = self._matrix_to_mob_matrix(matrix)
 self._organize_mob_matrix(mob_matrix)
 self.elements = VGroup(*it.chain(*mob_matrix))
 self.add(self.elements)
 self._add_brackets(self.left_bracket, self.right_bracket, **bracket_config)
 self.center()
 self.mob_matrix = mob_matrix
 if self.add_background_rectangles_to_entries:
 for mob in self.elements:
 mob.add_background_rectangle()
 if self.include_background_rectangle:
 self.add_background_rectangle()

def _matrix_to_mob_matrix(self, matrix):
 return [
 [
 self.element_to_mob(item, **self.element_to_mob_config)
 for item in row
]
 for row in matrix
]

def _organize_mob_matrix(self, matrix):
 for i, row in enumerate(matrix):
 for j, _ in enumerate(row):
 mob = matrix[i][j]
 mob.move_to(
 i * self.v_buff * DOWN + j * self.h_buff * RIGHT,
 self.element_alignment_corner,
)
 return self

def _add_brackets(self, left: str = "[", right: str = "]", **kwargs):
 """Adds the brackets to the Matrix mobject.

 See Latex document for various bracket types.

 Parameters

 left
 the left bracket, by default "["
 right
 the right bracket, by default "]"

 Returns

 :class:`Matrix`
 The current matrix object (self).
 """
 # Height per row of LaTeX array with default settings
 BRACKET_HEIGHT = 0.5977

 n = int((self.height) / BRACKET_HEIGHT) + 1
 empty_tex_array = "".join(
 [
 r"\begin{array}{c}",
 *n * [r"\quad \\"],
 r"\end{array}",
]
)

```

```

tex_left = """.join(
 [
 r"\left" + left,
 empty_tex_array,
 r"\right.",
]
)
tex_right = """.join(
 [
 r"\left.",
 empty_tex_array,
 r"\right" + right,
]
)
l_bracket = MathTex(tex_left, **kwargs)
r_bracket = MathTex(tex_right, **kwargs)

bracket_pair = VGroup(l_bracket, r_bracket)
if self.stretch_brackets:
 bracket_pair.stretch_to_fit_height(self.height + 2 * self.bracket_v_buff)
l_bracket.next_to(self, LEFT, self.bracket_h_buff)
r_bracket.next_to(self, RIGHT, self.bracket_h_buff)
self.brackets = bracket_pair
self.add(l_bracket, r_bracket)
return self

def get_columns(self):
 """Return columns of the matrix as VGroups.

 Returns

 List[:class:`~.VGroup`]
 Each VGroup contains a column of the matrix.

 Examples

 .. manim:: GetColumnsExample
 :save_last_frame:

 class GetColumnsExample(Scene):
 def construct(self):
 m0 = Matrix([["\pi", 3], [1, 5]])
 m0.add(SurroundingRectangle(m0.get_columns()[1]))
 self.add(m0)

 """
 return VGroup(
 (VGroup([row[i] for row in self.mob_matrix])
 for i in range(len(self.mob_matrix[0])))
)
)

def set_column_colors(self, *colors: str):
 """Set individual colors for each columns of the matrix.

 Parameters

 colors
 The list of colors; each color specified corresponds to a column.

 Returns

 :class:`Matrix`
 The current matrix object (self).

 Examples

 .. manim:: SetColumnColorsExample
 :save_last_frame:

 class SetColumnColorsExample(Scene):
 def construct(self):
 m0 = Matrix([["\pi", 1], [-1, 3]],
).set_column_colors([RED, BLUE], GREEN)
 self.add(m0)

 """
 columns = self.get_columns()
 for color, column in zip(colors, columns):
 column.set_color(color)
 return self

def get_rows(self):
 """Return rows of the matrix as VGroups.

```

```

 Returns

List[:class:`~.VGroup`]
 Each VGroup contains a row of the matrix.

 Examples

.. manim:: GetRowsExample
 :save_last_frame:

 class GetRowsExample(Scene):
 def construct(self):
 m0 = Matrix([["\\pi", 3], [1, 5]])
 m0.add(SurroundingRectangle(m0.get_rows()[1]))
 self.add(m0)
 """
 return VGroup(*[VGroup(*row) for row in self.mob_matrix])

def set_row_colors(self, *colors: str):
 """Set individual colors for each row of the matrix.

 Parameters

 colors
 The list of colors; each color specified corresponds to a row.

 Returns

:class:`Matrix`
 The current matrix object (self).

 Examples

.. manim:: SetRowColorsExample
 :save_last_frame:

 class SetRowColorsExample(Scene):
 def construct(self):
 m0 = Matrix([["\\pi", 1], [-1, 3]]).set_row_colors([RED,BLUE], GREEN)
 self.add(m0)
 """
 rows = self.get_rows()
 for color, row in zip(colors, rows):
 row.set_color(color)
 return self

def add_background_to_entries(self):
 """Add a black background rectangle to the matrix,
 see above for an example.

 Returns

:class:`Matrix`
 The current matrix object (self).

 """
 for mob in self.get_entries():
 mob.add_background_rectangle()
 return self

def get_mob_matrix(self):
 """Return the underlying mob matrix mobjects.

 Returns

List[:class:`~.VGroup`]
 Each VGroup contains a row of the matrix.

 """
 return self.mob_matrix

def get_entries(self):
 """Return the individual entries of the matrix.

 Returns

:class:`~.VGroup`
 VGroup containing entries of the matrix.

 Examples

.. manim:: GetEntriesExample
 :save_last_frame:

 class GetEntriesExample(Scene):

```

```

 def construct(self):
 m0 = Matrix([[2, 3], [1, 5]])
 ent = m0.get_entries()
 colors = [BLUE, GREEN, YELLOW, RED]
 for k in range(len(colors)):
 ent[k].set_color(colors[k])
 self.add(m0)
 """
 return self.elements

def get_brackets(self):
 """Return the bracket mobjects.

 Returns

 List[:class:`~.VGroup`]
 Each VGroup contains a bracket

 Examples

 .. manim:: GetBracketsExample
 :save_last_frame:

 class GetBracketsExample(Scene):
 def construct(self):
 m0 = Matrix([["\pi", 3], [1, 5]])
 bra = m0.get_brackets()
 colors = [BLUE, GREEN]
 for k in range(len(colors)):
 bra[k].set_color(colors[k])
 self.add(m0)
 """
 return self.brackets

class DecimalMatrix(Matrix):
 """A mobject that displays a matrix with decimal entries on the screen.

 Examples

 .. manim:: DecimalMatrixExample
 :save_last_frame:

 class DecimalMatrixExample(Scene):
 def construct(self):
 m0 = DecimalMatrix(
 [[3.456, 2.122], [33.2244, 12]],
 element_to_mobject_config={"num_decimal_places": 2},
 left_bracket="\{",
 right_bracket="\}")
 self.add(m0)
 """

 def __init__(
 self,
 matrix: Iterable,
 element_to_mobject: Mobject = DecimalNumber,
 element_to_mobject_config: dict[str, Mobject] = {"num_decimal_places": 1},
 **kwargs,
):
 """
 Will round/truncate the decimal places as per the provided config.

 Parameters

 matrix
 A numpy 2d array or list of lists
 element_to_mobject
 Mobject to use, by default DecimalNumber
 element_to_mobject_config
 Config for the desired mobject, by default {"num_decimal_places": 1}
 """
 super().__init__(
 matrix,
 element_to_mobject=element_to_mobject,
 element_to_mobject_config=element_to_mobject_config,
 **kwargs,
)

 class IntegerMatrix(Matrix):
 """A mobject that displays a matrix with integer entries on the screen.

 Examples

```

```

.. manim:: IntegerMatrixExample
 :save_last_frame:

 class IntegerMatrixExample(Scene):
 def construct(self):
 m0 = IntegerMatrix(
 [[3.7, 2], [42.2, 12]],
 left_bracket="(",
 right_bracket=")")
 self.add(m0)
 """
 def __init__(self, matrix: Iterable, element_to_mobject: Mobject = Integer, **kwargs):
 """
 Will round if there are decimal entries in the matrix.

 Parameters

 matrix
 A numpy 2d array or list of lists
 element_to_mobject
 Mobject to use, by default Integer
 """
 super().__init__(matrix, element_to_mobject=element_to_mobject, **kwargs)

class MobjectMatrix(Matrix):
 """A mobject that displays a matrix of mobject entries on the screen.

Examples

.. manim:: MobjectMatrixExample
 :save_last_frame:

 class MobjectMatrixExample(Scene):
 def construct(self):
 a = Circle().scale(0.3)
 b = Square().scale(0.3)
 c = MathTex("\pi").scale(2)
 d = Star().scale(0.3)
 m0 = MobjectMatrix([[a, b], [c, d]])
 self.add(m0)
 """
 def __init__(self, matrix, element_to_mobject=lambda m: m, **kwargs):
 super().__init__(matrix, element_to_mobject=element_to_mobject, **kwargs)

def get_det_text(
 matrix: Matrix,
 determinant: int | str | None = None,
 background_rect: bool = False,
 initial_scale_factor: float = 2,
):
 """
 Helper function to create determinant.

 Parameters

 matrix
 The matrix whose determinant is to be created

 determinant
 The value of the determinant of the matrix

 background_rect
 The background rectangle

 initial_scale_factor
 The scale of the text `det` w.r.t the matrix

 Returns

 :class:`~.VGroup`
 A VGroup containing the determinant

Examples

.. manim:: DeterminantOfAMatrix
 :save_last_frame:

 class DeterminantOfAMatrix(Scene):
 def construct(self):

```

```

matrix = Matrix([
 [2, 0],
 [-1, 1]
])

scaling down the `det` string
det = get_det_text(matrix,
 determinant=3,
 initial_scale_factor=1)

must add the matrix
self.add(matrix)
self.add(det)

"""
parens = MathTex("(", ")")
parens.scale(initial_scale_factor)
parens.stretch_to_fit_height(matrix.height)
l_paren, r_paren = parens.split()
l_paren.next_to(matrix, LEFT, buff=0.1)
r_paren.next_to(matrix, RIGHT, buff=0.1)
det = Tex("det")
det.scale(initial_scale_factor)
det.next_to(l_paren, LEFT, buff=0.1)
if background_rect:
 det.add_background_rectangle()
det_text = VGroup(det, l_paren, r_paren)
if determinant is not None:
 eq = MathTex("=")
 eq.next_to(r_paren, RIGHT, buff=0.1)
 result = MathTex(str(determinant))
 result.next_to(eq, RIGHT, buff=0.2)
 det_text.add(eq, result)
return det_text

```

# communitycolors.py

```
"""The colormap of manim community"""

from __future__ import annotations
```

```
LOGO_WHITE = "#ece6e2"
LOGO_GREEN = "#87c2a5"
LOGO_BLUE = "#525893"
LOGO_RED = "#e07a5f"
LOGO_BLACK = "#343434"
```

# rate\_functions.py

```
"""A selection of rate functions, i.e., *speed curves* for animations.
Please find a standard list at https://easings.net/. Here is a picture
for the non-standard ones

.. manim:: RateFuncExample
 :save_last_frame:

class RateFuncExample(Scene):
 def construct(self):
 x = VGroup()
 for k, v in rate_functions.__dict__.items():
 if "function" in str(v):
 if (
 not k.startswith("__")
 and not k.startswith("sqrt")
 and not k.startswith("bezier")
):
 try:
 rate_func = v
 plot = (
 ParametricFunction(
 lambda x: [x, rate_func(x), 0],
 t_range=[0, 1, .01],
 use_smoothing=False,
 color=YELLOW,
)
 .stretch_to_fit_width(1.5)
 .stretch_to_fit_height(1)
)
 plot_bg = SurroundingRectangle(plot).set_color(WHITE)
 plot_title = (
 Text(rate_func.__name__, weight=BOLD)
 .scale(0.5)
 .next_to(plot_bg, UP, buff=0.1)
)
 x.add(VGroup(plot_bg, plot, plot_title))
 except: # because functions `not Quite There`, `function squish_rate_func` are not working.
 pass
 x.arrange_in_grid(cols=8)
 x.height = config.frame_height
 x.width = config.frame_width
 x.move_to(ORIGIN).scale(0.95)
 self.add(x)
```

There are primarily 3 kinds of standard easing functions:

```
#. Ease In - The animation has a smooth start.
#. Ease Out - The animation has a smooth end.
#. Ease In Out - The animation has a smooth start as well as smooth end.

.. note:: The standard functions are not exported, so to use them you do something like this:
 rate_func=rate_functions.ease_in_sine
 On the other hand, the non-standard functions, which are used more commonly, are exported and can be used directly.
```

```
.. manim:: RateFunctions1Example

class RateFunctions1Example(Scene):
 def construct(self):
 line1 = Line(3*LEFT, 3*RIGHT).shift(UP).set_color(RED)
 line2 = Line(3*LEFT, 3*RIGHT).set_color(GREEN)
 line3 = Line(3*LEFT, 3*RIGHT).shift(DOWN).set_color(BLUE)

 dot1 = Dot().move_to(line1.get_left())
 dot2 = Dot().move_to(line2.get_left())
 dot3 = Dot().move_to(line3.get_left())

 label1 = Tex("Ease In").next_to(line1, RIGHT)
 label2 = Tex("Ease out").next_to(line2, RIGHT)
 label3 = Tex("Ease In Out").next_to(line3, RIGHT)

 self.play(
 FadeIn(VGroup(line1, line2, line3)),
 FadeIn(VGroup(dot1, dot2, dot3)),
 Write(VGroup(label1, label2, label3)),
)
 self.play(
 MoveAlongPath(dot1, line1, rate_func=rate_functions.ease_in_sine),
 MoveAlongPath(dot2, line2, rate_func=rate_functions.ease_out_sine),
 MoveAlongPath(dot3, line3, rate_func=rate_functions.ease_in_out_sine),
 run_time=7
)
 self.wait()
```

"""

```

from __future__ import annotations

__all__ = [
 "linear",
 "smooth",
 "rush_into",
 "rush_from",
 "slow_into",
 "double_smooth",
 "there_and_back",
 "there_and_back_with_pause",
 "running_start",
 "not Quite there",
 "wiggle",
 "squish_rate_func",
 "lingering",
 "exponential_decay",
]
]

import typing
from functools import wraps
from math import sqrt

import numpy as np

from ..utils.bezier import bezier
from ..utils.simple_functions import sigmoid

This is a decorator that makes sure any function it's used on will
return 0 if t<0 and 1 if t>1.
def unit_interval(function):
 @wraps(function)
 def wrapper(t, *args, **kwargs):
 if 0 <= t <= 1:
 return function(t, *args, **kwargs)
 elif t < 0:
 return 0
 else:
 return 1
 return wrapper

This is a decorator that makes sure any function it's used on will
return 0 if t<0 or t>1.
def zero(function):
 @wraps(function)
 def wrapper(t, *args, **kwargs):
 if 0 <= t <= 1:
 return function(t, *args, **kwargs)
 else:
 return 0
 return wrapper

@unit_interval
def linear(t: float) -> float:
 return t

@unit_interval
def smooth(t: float, inflection: float = 10.0) -> float:
 error = sigmoid(-inflection / 2)
 return min(
 max((sigmoid(inflection * (t - 0.5)) - error) / (1 - 2 * error), 0),
 1,
)

@unit_interval
def rush_into(t: float, inflection: float = 10.0) -> float:
 return 2 * smooth(t / 2.0, inflection)

@unit_interval
def rush_from(t: float, inflection: float = 10.0) -> float:
 return 2 * smooth(t / 2.0 + 0.5, inflection) - 1

@unit_interval
def slow_into(t: float) -> float:
 return np.sqrt(1 - (1 - t) * (1 - t))

@unit_interval
def double_smooth(t: float) -> float:
 if t < 0.5:
 return 0.5 * smooth(2 * t)

```

```

else:
 return 0.5 * (1 + smooth(2 * t - 1))

@zero
def there_and_back(t: float, inflection: float = 10.0) -> float:
 new_t = 2 * t if t < 0.5 else 2 * (1 - t)
 return smooth(new_t, inflection)

@zero
def there_and_back_with_pause(t: float, pause_ratio: float = 1.0 / 3) -> float:
 a = 1.0 / pause_ratio
 if t < 0.5 - pause_ratio / 2:
 return smooth(a * t)
 elif t < 0.5 + pause_ratio / 2:
 return 1
 else:
 return smooth(a - a * t)

@unit_interval
def running_start(
 t: float,
 pull_factor: float = -0.5,
) -> typing.Iterable: # what is func return type?
 return bezier([0, 0, pull_factor, pull_factor, 1, 1])(t)

def not Quite There(
 func: typing.Callable[[float], float] = smooth,
 proportion: float = 0.7,
) -> typing.Callable[[float], float]:
 def result(t):
 return proportion * func(t)

 return result

@zero
def wiggle(t: float, wiggles: float = 2) -> float:
 return there_and_back(t) * np.sin(wiggles * np.pi * t)

def squish_rate_func(
 func: typing.Callable[[float], float],
 a: float = 0.4,
 b: float = 0.6,
) -> typing.Callable[[float], float]:
 def result(t):
 if a == b:
 return a

 if t < a:
 return func(0)
 elif t > b:
 return func(1)
 else:
 return func((t - a) / (b - a))

 return result

Stylistically, should this take parameters (with default values)?
Ultimately, the functionality is entirely subsumed by squish_rate_func,
but it may be useful to have a nice name for with nice default params for
"lingering", different from squish_rate_func's default params

@unit_interval
def lingering(t: float) -> float:
 return squish_rate_func(lambda t: t, 0, 0.8)(t)

@unit_interval
def exponential_decay(t: float, half_life: float = 0.1) -> float:
 # The half-life should be rather small to minimize
 # the cut-off error at the end
 return 1 - np.exp(-t / half_life)

@unit_interval
def ease_in_sine(t: float) -> float:
 return 1 - np.cos((t * np.pi) / 2)

@unit_interval
def ease_out_sine(t: float) -> float:
 return np.sin((t * np.pi) / 2)

```

```

@unit_interval
def ease_in_out_sine(t: float) -> float:
 return -(np.cos(np.pi * t) - 1) / 2

@unit_interval
def ease_in_quad(t: float) -> float:
 return t * t

@unit_interval
def ease_out_quad(t: float) -> float:
 return 1 - (1 - t) * (1 - t)

@unit_interval
def ease_in_out_quad(t: float) -> float:
 return 2 * t * t if t < 0.5 else 1 - pow(-2 * t + 2, 2) / 2

@unit_interval
def ease_in_cubic(t: float) -> float:
 return t * t * t

@unit_interval
def ease_out_cubic(t: float) -> float:
 return 1 - pow(1 - t, 3)

@unit_interval
def ease_in_out_cubic(t: float) -> float:
 return 4 * t * t * t if t < 0.5 else 1 - pow(-2 * t + 2, 3) / 2

@unit_interval
def ease_in_quart(t: float) -> float:
 return t * t * t * t

@unit_interval
def ease_out_quart(t: float) -> float:
 return 1 - pow(1 - t, 4)

@unit_interval
def ease_in_out_quart(t: float) -> float:
 return 8 * t * t * t * t if t < 0.5 else 1 - pow(-2 * t + 2, 4) / 2

@unit_interval
def ease_in_quint(t: float) -> float:
 return t * t * t * t * t

@unit_interval
def ease_out_quint(t: float) -> float:
 return 1 - pow(1 - t, 5)

@unit_interval
def ease_in_out_quint(t: float) -> float:
 return 16 * t * t * t * t * t if t < 0.5 else 1 - pow(-2 * t + 2, 5) / 2

@unit_interval
def ease_in_expo(t: float) -> float:
 return 0 if t == 0 else pow(2, 10 * t - 10)

@unit_interval
def ease_out_expo(t: float) -> float:
 return 1 if t == 1 else 1 - pow(2, -10 * t)

@unit_interval
def ease_in_out_expo(t: float) -> float:
 if t == 0:
 return 0
 elif t == 1:
 return 1
 elif t < 0.5:
 return pow(2, 20 * t - 10) / 2
 else:
 return (2 - pow(2, -20 * t + 10)) / 2

@unit_interval
def ease_in_circ(t: float) -> float:
 return 1 - sqrt(1 - pow(t, 2))

```

```

@unit_interval
def ease_out_circ(t: float) -> float:
 return sqrt(1 - pow(t - 1, 2))

@unit_interval
def ease_in_out_circ(t: float) -> float:
 return (
 (1 - sqrt(1 - pow(2 * t, 2))) / 2
 if t < 0.5
 else (sqrt(1 - pow(-2 * t + 2, 2)) + 1) / 2
)

@unit_interval
def ease_in_back(t: float) -> float:
 c1 = 1.70158
 c3 = c1 + 1
 return c3 * t * t * t - c1 * t * t

@unit_interval
def ease_out_back(t: float) -> float:
 c1 = 1.70158
 c3 = c1 + 1
 return 1 + c3 * pow(t - 1, 3) + c1 * pow(t - 1, 2)

@unit_interval
def ease_in_out_back(t: float) -> float:
 c1 = 1.70158
 c2 = c1 * 1.525
 return (
 (pow(2 * t, 2) * ((c2 + 1) * 2 * t - c2)) / 2
 if t < 0.5
 else (pow(2 * t - 2, 2) * ((c2 + 1) * (t * 2 - 2) + c2) + 2) / 2
)

@unit_interval
def ease_in_elastic(t: float) -> float:
 c4 = (2 * np.pi) / 3
 if t == 0:
 return 0
 elif t == 1:
 return 1
 else:
 return -pow(2, 10 * t - 10) * np.sin((t * 10 - 10.75) * c4)

@unit_interval
def ease_out_elastic(t: float) -> float:
 c4 = (2 * np.pi) / 3
 if t == 0:
 return 0
 elif t == 1:
 return 1
 else:
 return pow(2, -10 * t) * np.sin((t * 10 - 0.75) * c4) + 1

@unit_interval
def ease_in_out_elastic(t: float) -> float:
 c5 = (2 * np.pi) / 4.5
 if t == 0:
 return 0
 elif t == 1:
 return 1
 elif t < 0.5:
 return -(pow(2, 20 * t - 10) * np.sin((20 * t - 11.125) * c5)) / 2
 else:
 return (pow(2, -20 * t + 10) * np.sin((20 * t - 11.125) * c5)) / 2 + 1

@unit_interval
def ease_in_bounce(t: float) -> float:
 return 1 - ease_out_bounce(1 - t)

@unit_interval
def ease_out_bounce(t: float) -> float:
 n1 = 7.5625
 d1 = 2.75

 if t < 1 / d1:
 return n1 * t * t
 elif t < 2 / d1:
 return n1 * (t - 1.5 / d1) * (t - 1.5 / d1) + 0.75
 elif t < 2.5 / d1:
 return n1 * (t - 2.25 / d1) * (t - 2.25 / d1) + 0.9375

```

```
else:
 return n1 * (t - 2.625 / d1) * (t - 2.625 / d1) + 0.984375

@unit_interval
def ease_in_out_bounce(t: float) -> float:
 if t < 0.5:
 return (1 - ease_out_bounce(1 - 2 * t)) / 2
 else:
 return (1 + ease_out_bounce(2 * t - 1)) / 2
```

# family.py

```
from __future__ import annotations

import itertools as it
from typing import Iterable

from ..mobject.mobject import Mobject
from ..utils.iterables import remove_list_redundancies

def extract_mobject_family_members(
 mobjects: Iterable[Mobject],
 use_z_index=False,
 only_those_with_points: bool = False,
):
 """Returns a list of the types of mobjects and their family members present.
 A "family" in this context refers to a mobject, its submobjects, and their
 submobjects, recursively.

 Parameters

 mobjects
 The Mobjects currently in the Scene
 only_those_with_points
 Whether or not to only do this for
 those mobjects that have points. By default False

 Returns

 list
 list of the mobjects and family members.
 """
 if only_those_with_points:
 method = Mobject.family_members_with_points
 else:
 method = Mobject.get_family
 extracted_mobjects = remove_list_redundancies(
 list(it.chain(*(method(m) for m in mobjects))),
)
 if use_z_index:
 return sorted(extracted_mobjects, key=lambda m: m.z_index)
 return extracted_mobjects
```

# three\_dimensions.py

```
"""Three-dimensional mobjects."""
from __future__ import annotations

__all__ = [
 "ThreeDVMobject",
 "Surface",
 "Sphere",
 "Dot3D",
 "Cube",
 "Prism",
 "Cone",
 "Arrow3D",
 "Cylinder",
 "Line3D",
 "Torus",
]
from typing import TYPE_CHECKING

import numpy as np

from manim import config, logger
from manim.constants import *
from manim.mobject.geometry.arc import Circle
from manim.mobject.geometry.polygram import Square
from manim.mobject.mobject import *
from manim.mobject.opengl_compatibility import ConvertToOpenGL
from manim.mobject.opengl_mobject import OpenGLMobject
from manim.mobject.types.vectorized_mobject import VGroup, VMobject
from manim.utils.color import *
from manim.utils.iterables import tuplify
from manim.utils.space_ops import normalize, perpendicular_bisector, z_to_vector

if TYPE_CHECKING:
 from typing import *

 from colour import Color

class ThreeDVMobject(VMobject, metaclass=ConvertToOpenGL):
 def __init__(self, shade_in_3d: bool = True, **kwargs):
 super().__init__(shade_in_3d=shade_in_3d, **kwargs)

class Surface(VGroup, metaclass=ConvertToOpenGL):
 """Creates a Parametric Surface using a checkerboard pattern.

 Parameters

 func
 The function defining the :class:`Surface`.
 u_range
 The range of the ``u`` variable: ``(``u_min, u_max``)``.
 v_range
 The range of the ``v`` variable: ``(``v_min, v_max``)``.
 resolution
 The number of samples taken of the :class:`Surface`. A tuple can be used to define different resolutions for ``u`` and ``v`` respectively.
 fill_color
 The color of the :class:`Surface`. Ignored if ``checkerboard_colors`` is set.
 fill_opacity
 The opacity of the :class:`Surface`, from 0 being fully transparent to 1 being fully opaque. Defaults to 1.
 checkerboard_colors
 ng individual faces alternating colors. Overrides ``fill_color``.
 stroke_color
 Color of the stroke surrounding each face of :class:`Surface`.
 stroke_width
 Width of the stroke surrounding each face of :class:`Surface`. Defaults to 0.5.
 should_make_jagged
 Changes the anchor mode of the BÃ©zier curves from smooth to jagged. Defaults to ``False``.

 Examples

 .. manim:: ParaSurface
 :save_last_frame:

 class ParaSurface(ThreeDScene):
 def func(self, u, v):
 return np.array([np.cos(u) * np.cos(v), np.cos(u) * np.sin(v), u])

 def construct(self):
 axes = ThreeDAxes(x_range=[-4, 4], x_length=8)
 surface = Surface(

```

```

 lambda u, v: axes.c2p(*self.func(u, v)),
 u_range=[-PI, PI],
 v_range=[0, TAU],
 resolution=8,
)
 self.set_camera_orientation(theta=70 * DEGREES, phi=75 * DEGREES)
 self.add(axes, surface)
"""

def __init__(
 self,
 func: Callable[[float, float], np.ndarray],
 u_range: Sequence[float] = [0, 1],
 v_range: Sequence[float] = [0, 1],
 resolution: Sequence[int] = 32,
 surface_piece_config: dict = {},
 fill_color: Color = BLUE_D,
 fill_opacity: float = 1.0,
 checkerboard_colors: Sequence[Color] = [BLUE_D, BLUE_E],
 stroke_color: Color = LIGHT_GREY,
 stroke_width: float = 0.5,
 should_make_jagged: bool = False,
 pre_function_handle_to_anchor_scale_factor: float = 0.00001,
 **kwargs,
) -> None:
 self.u_range = u_range
 self.v_range = v_range
 super().__init__(**kwargs)
 self.resolution = resolution
 self.surface_piece_config = surface_piece_config
 self.fill_color = fill_color
 self.fill_opacity = fill_opacity
 self.checkerboard_colors = checkerboard_colors
 self.stroke_color = stroke_color
 self.stroke_width = stroke_width
 self.should_make_jagged = should_make_jagged
 self.pre_function_handle_to_anchor_scale_factor = (
 pre_function_handle_to_anchor_scale_factor
)
 self._func = func
 self._setup_in_uv_space()
 self.apply_function(lambda p: func(p[0], p[1]))
 if self.should_make_jagged:
 self.make_jagged()

def func(self, u: float, v: float) -> np.ndarray:
 """The z values defining the :class:`Surface` being plotted.

 Returns

 :class:`numpy.array`
 The z values defining the :class:`Surface`.
 """
 return self._func(u, v)

def _get_u_values_and_v_values(self):
 res = tuplify(self.resolution)
 if len(res) == 1:
 u_res = v_res = res[0]
 else:
 u_res, v_res = res

 u_values = np.linspace(*self.u_range, u_res + 1)
 v_values = np.linspace(*self.v_range, v_res + 1)

 return u_values, v_values

def _setup_in_uv_space(self):
 u_values, v_values = self._get_u_values_and_v_values()
 faces = VGroup()
 for i in range(len(u_values) - 1):
 for j in range(len(v_values) - 1):
 u1, u2 = u_values[i : i + 2]
 v1, v2 = v_values[j : j + 2]
 face = ThreeDVMobject()
 face.set_points_as_corners(
 [
 [u1, v1, 0],
 [u2, v1, 0],
 [u2, v2, 0],
 [u1, v2, 0],
 [u1, v1, 0],
],
)
 faces.add(face)
 face.u_index = i
 face.v_index = j
 face.ul = u1
 face.u2 = u2
 face.v1 = v1
 face.v2 = v2
 faces.set_fill(color=self.fill_color, opacity=self.fill_opacity)
 faces.set_stroke(
 color=self.stroke_color,

```

```

 width=self.stroke_width,
 opacity=self.stroke_opacity,
)
 self.add(*faces)
 if self.checkerboard_colors:
 self.set_fill_by_checkerboard(*self.checkerboard_colors)

def set_fill_by_checkerboard(
 self, *colors: Sequence[Color], opacity: float = None
) -> Mobject:
 """Sets the fill_color of each face of :class:`Surface` in
 an alternating pattern.

 Parameters

 colors
 List of colors for alternating pattern.
 opacity
 The fill_opacity of :class:`Surface`, from 0 being fully transparent
 to 1 being fully opaque.

 Returns

 :class:`~.Surface`
 The parametric surface with an alternating pattern.
 """
 n_colors = len(colors)
 for face in self:
 c_index = (face.u_index + face.v_index) % n_colors
 face.set_fill(colors[c_index], opacity=opacity)
 return self

def set_fill_by_value(
 self,
 axes: Mobject,
 colorscale: Union[Iterable[Color], Color] | None = None,
 axis: int = 2,
 **kwargs,
) -> Mobject:
 """Sets the color of each mobject of a parametric surface to a color
 relative to its axis-value.

 Parameters

 axes
 The axes for the parametric surface, which will be used to map
 axis-values to colors.
 colorscale
 A list of colors, ordered from lower axis-values to higher axis-values.
 If a list of tuples is passed containing colors paired with numbers,
 then those numbers will be used as the pivots.
 axis
 The chosen axis to use for the color mapping. (0 = x, 1 = y, 2 = z)

 Returns

 :class:`~.Surface`
 The parametric surface with a gradient applied by value. For chaining.

 Examples

 .. manim:: FillByValueExample
 :save_last_frame:

 class FillByValueExample(ThreeDScene):
 def construct(self):
 resolution_fa = 8
 self.set_camera_orientation(phi=75 * DEGREES, theta=-160 * DEGREES)
 axes = ThreeDAxes(x_range=(0, 5, 1), y_range=(0, 5, 1), z_range=(-1, 1, 0.5))
 def param_surface(u, v):
 x = u
 y = v
 z = np.sin(x) * np.cos(y)
 return z
 surface_plane = Surface(
 lambda u, v: axes.c2p(u, v, param_surface(u, v)),
 resolution=(resolution_fa, resolution_fa),
 v_range=[0, 5],
 u_range=[0, 5],
)
 surface_plane.set_style(fill_opacity=1)
 surface_plane.set_fill_by_value(axes=axes, colorscale=[(RED, -0.5), (YELLOW, 0), (GREEN, 0.5)], axis=2)
 self.add(axes, surface_plane)
 """
 if "colors" in kwargs and colorscale is None:
 colorscale = kwargs.pop("colors")
 if kwargs:
 raise ValueError(
 "Unsupported keyword argument(s): "
 f"{''.join(str(key) for key in kwargs)}"
)
 if colorscale is None:
 logger.warning(
 "The value passed to the colorscale keyword argument was None, "

```

```

 "the surface fill color has not been changed"
)
 return self

ranges = [axes.x_range, axes.y_range, axes.z_range]

if type(colorscale[0]) is tuple:
 new_colors, pivots = [
 [i for i, j in colorscale],
 [j for i, j in colorscale],
]
else:
 new_colors = colorscale

pivot_min = ranges[axis][0]
pivot_max = ranges[axis][1]
pivot_frequency = (pivot_max - pivot_min) / (len(new_colors) - 1)
pivots = np.arange(
 start=pivot_min,
 stop=pivot_max + pivot_frequency,
 step=pivot_frequency,
)

for mob in self.family_members_with_points():
 axis_value = axes.point_to_coords(mob.get_midpoint())[axis]
 if axis_value <= pivots[0]:
 mob.set_color(new_colors[0])
 elif axis_value >= pivots[-1]:
 mob.set_color(new_colors[-1])
 else:
 for i, pivot in enumerate(pivots):
 if pivot > axis_value:
 color_index = (axis_value - pivots[i - 1]) / (
 pivots[i] - pivots[i - 1]
)
 color_index = min(color_index, 1)
 mob_color = interpolate_color(
 new_colors[i - 1],
 new_colors[i],
 color_index,
)
 if config.renderer == RendererType.OPENGL:
 mob.set_color(mob_color, recurse=False)
 elif config.renderer == RendererType.CAIRO:
 mob.set_color(mob_color, family=False)
 break
 return self

Specific shapes

class Sphere(Surface):
 """A three-dimensional sphere.

Parameters

center
 Center of the :class:`Sphere`.
radius
 The radius of the :class:`Sphere`.
resolution
 The number of samples taken of the :class:`Sphere`. A tuple can be used
 to define different resolutions for ``u`` and ``v`` respectively.
u_range
 The range of the ``u`` variable: ``(``u_min, u_max````).
v_range
 The range of the ``v`` variable: ``(``v_min, v_max````.

Examples

.. manim:: ExampleSphere
 :save_last_frame:

 class ExampleSphere(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=PI / 6, theta=PI / 6)
 sphere1 = Sphere(
 center=(3, 0, 0),
 radius=1,
 resolution=(20, 20),
 u_range=[0.001, PI - 0.001],
 v_range=[0, TAU]
)
 sphere1.set_color(RED)
 self.add(sphere1)
 sphere2 = Sphere(center=(-1, -3, 0), radius=2, resolution=(18, 18))
 sphere2.set_color(GREEN)
 self.add(sphere2)
 sphere3 = Sphere(center=(-1, 2, 0), radius=2, resolution=(16, 16))
 sphere3.set_color(BLUE)
 self.add(sphere3)

```

```

"""
def __init__(
 self,
 center: Sequence[float] = ORIGIN,
 radius: float = 1,
 resolution: Sequence[int] = None,
 u_range: Sequence[float] = (0, TAU),
 v_range: Sequence[float] = (0, PI),
 **kwargs,
) -> None:
 if config.renderer == RendererType.OPENGL:
 res_value = (101, 51)
 elif config.renderer == RendererType.CAIRO:
 res_value = (24, 12)
 else:
 raise Exception("Unknown renderer")

 resolution = resolution if resolution is not None else res_value

 self.radius = radius

 super().__init__(
 self.func,
 resolution=resolution,
 u_range=u_range,
 v_range=v_range,
 **kwargs,
)

 self.shift(center)

def func(self, u: float, v: float) -> np.ndarray:
 """The z values defining the :class:`Sphere` being plotted.

 Returns

 :class:`numpy.array`
 The z values defining the :class:`Sphere`.

 """
 return self.radius * np.array(
 [np.cos(u) * np.sin(v), np.sin(u) * np.sin(v), -np.cos(v)],
)
}

class Dot3D(Sphere):
 """A spherical dot.

 Parameters

 point
 The location of the dot.
 radius
 The radius of the dot.
 color
 The color of the :class:`Dot3D`.
 resolution
 The number of samples taken of the :class:`Dot3D`. A tuple can be
 used to define different resolutions for ``u`` and ``v`` respectively.

 Examples

 .. manim:: Dot3DExample
 :save_last_frame:

 class Dot3DExample(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75*DEGREES, theta=-45*DEGREES)

 axes = ThreeDAxes()
 dot_1 = Dot3D(point=axes.coords_to_point(0, 0, 1), color=RED)
 dot_2 = Dot3D(point=axes.coords_to_point(2, 0, 0), radius=0.1, color=BLUE)
 dot_3 = Dot3D(point=[0, 0, 0], radius=0.1, color=ORANGE)
 self.add(axes, dot_1, dot_2, dot_3)
 """

 def __init__(
 self,
 point: list | np.ndarray = ORIGIN,
 radius: float = DEFAULT_DOT_RADIUS,
 color: Color = WHITE,
 resolution: Sequence[int] = (8, 8),
 **kwargs,
) -> None:
 super().__init__(center=point, radius=radius, resolution=resolution, **kwargs)
 self.set_color(color)

class Cube(VGroup):
 """A three-dimensional cube.

 Parameters

```

```

side_length
 Length of each side of the :class:`Cube`.
fill_opacity
 The opacity of the :class:`Cube`, from 0 being fully transparent to 1 being
 fully opaque. Defaults to 0.75.
fill_color
 The color of the :class:`Cube`.
stroke_width
 The width of the stroke surrounding each face of the :class:`Cube`.

Examples

.. manim:: CubeExample
 :save_last_frame:

 class CubeExample(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=75*DEGREES, theta=-45*DEGREES)

 axes = ThreeDAxes()
 cube = Cube(side_length=3, fill_opacity=0.7, fill_color=BLUE)
 self.add(cube)

 """"

def __init__(self,
 side_length: float = 2,
 fill_opacity: float = 0.75,
 fill_color: Color = BLUE,
 stroke_width: float = 0,
 **kwargs,
) -> None:
 self.side_length = side_length
 super().__init__(
 fill_color=fill_color,
 fill_opacity=fill_opacity,
 stroke_width=stroke_width,
 **kwargs,
)

def generate_points(self) -> None:
 """Creates the sides of the :class:`Cube`."""
 for vect in IN, OUT, LEFT, RIGHT, UP, DOWN:
 face = Square(
 side_length=self.side_length,
 shade_in_3d=True,
)
 face.flip()
 face.shift(self.side_length * OUT / 2.0)
 face.apply_matrix(z_to_vector(vect))

 self.add(face)

init_points = generate_points

class Prism(Cube):
 """A right rectangular prism (or rectangular cuboid).
 Defined by the length of each side in ``[x, y, z]`` format.

Parameters

dimensions
 Dimensions of the :class:`Prism` in ``[x, y, z]`` format.

Examples

.. manim:: ExamplePrism
 :save_last_frame:

 class ExamplePrism(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(phi=60 * DEGREES, theta=150 * DEGREES)
 prismSmall = Prism(dimensions=[1, 2, 3]).rotate(PI / 2)
 prismLarge = Prism(dimensions=[1.5, 3, 4.5]).move_to([2, 0, 0])
 self.add(prismSmall, prismLarge)

 """"

def __init__(self, dimensions: Sequence[int] = [3, 2, 1], **kwargs) -> None:
 self.dimensions = dimensions
 super().__init__(**kwargs)

def generate_points(self) -> None:
 """Creates the sides of the :class:`Prism`."""
 super().generate_points()
 for dim, value in enumerate(self.dimensions):
 self.rescale_to_fit(value, dim, stretch=True)

class Cone(Surface):
 """A circular cone.
 Can be defined using 2 parameters: its height, and its base radius.

```

The polar angle, theta, can be calculated using `arctan(base_radius / height)`. The spherical radius, r, is calculated using the pythagorean theorem.

Parameters

-----

`base_radius`  
The base radius from which the cone tapers.

`height`  
The height measured from the plane formed by the `base_radius` to the apex of the cone.

`direction`  
The direction of the apex.

`show_base`  
Whether to show the base plane or not.

`v_range`  
The azimuthal angle to start and end at.

`u_min`  
The radius at the apex.

`checkerboard_colors`  
Show checkerboard grid texture on the cone.

Examples

-----

```
.. manim:: ExampleCone
 :save_last_frame:

 class ExampleCone(ThreeDScene):
 def construct(self):
 axes = ThreeDAxes()
 cone = Cone(direction=X_AXIS+Y_AXIS+2*Z_AXIS, resolution=8)
 self.set_camera_orientation(phi=5*PI/11, theta=PI/9)
 self.add(axes, cone)

 """
 def __init__(self,
 base_radius: float = 1,
 height: float = 1,
 direction: np.ndarray = Z_AXIS,
 show_base: bool = False,
 v_range: Sequence[float] = [0, TAU],
 u_min: float = 0,
 checkerboard_colors: bool = False,
 **kwargs,
):
 self.direction = direction
 self.theta = PI - np.arctan(base_radius / height)

 super().__init__(
 self_func,
 v_range=v_range,
 u_range=[u_min, np.sqrt(base_radius**2 + height**2)],
 checkerboard_colors=checkerboard_colors,
 **kwargs,
)
 # used for rotations
 self._current_theta = 0
 self._current_phi = 0

 if show_base:
 self.base_circle = Circle(
 radius=base_radius,
 color=self.fill_color,
 fill_opacity=self.fill_opacity,
 stroke_width=0,
)
 self.base_circle.shift(height * IN)
 self.add(self.base_circle)

 self._rotate_to_direction()

 def func(self, u: float, v: float) -> np.ndarray:
 """Converts from spherical coordinates to cartesian.

 Parameters

 u
 The radius.
 v
 The azimuthal angle.

 Returns

 :class:`numpy.array`
 Points defining the :class:`Cone`.

 """
 r = u
 phi = v
 return np.array([
 r * np.sin(self.theta) * np.cos(phi),
 r * np.sin(self.theta) * np.sin(phi),
 r * np.cos(self.theta),
])
```

```

],
)

def _rotate_to_direction(self):
 x, y, z = self.direction

 r = np.sqrt(x**2 + y**2 + z**2)
 if r > 0:
 theta = np.arccos(z / r)
 else:
 theta = 0

 if x == 0:
 if y == 0: # along the z axis
 phi = 0
 else:
 phi = np.arctan(np.inf)
 if y < 0:
 phi += PI
 else:
 phi = np.arctan(y / x)
 if x < 0:
 phi += PI

 # Undo old rotation (in reverse order)
 self.rotate(-self._current_phi, Z_AXIS, about_point=ORIGIN)
 self.rotate(-self._current_theta, Y_AXIS, about_point=ORIGIN)

 # Do new rotation
 self.rotate(theta, Y_AXIS, about_point=ORIGIN)
 self.rotate(phi, Z_AXIS, about_point=ORIGIN)

 # Store values
 self._current_theta = theta
 self._current_phi = phi

def set_direction(self, direction: np.ndarray) -> None:
 """Changes the direction of the apex of the :class:`Cone`.

 Parameters

 direction
 The direction of the apex.
 """
 self.direction = direction
 self._rotate_to_direction()

def get_direction(self) -> np.ndarray:
 """Returns the current direction of the apex of the :class:`Cone`.

 Returns

 direction : :class:`numpy.array`
 The direction of the apex.
 """
 return self.direction

class Cylinder(Surface):
 """A cylinder, defined by its height, radius and direction,

 Parameters

 radius
 The radius of the cylinder.
 height
 The height of the cylinder.
 direction
 The direction of the central axis of the cylinder.
 v_range
 The height along the height axis (given by direction) to start and end on.
 show_ends
 Whether to show the end caps or not.
 resolution
 The number of samples taken of the :class:`Cylinder`. A tuple can be used
 to define different resolutions for ``u`` and ``v`` respectively.

 Examples

 .. manim:: ExampleCylinder
 :save_last_frame:

 class ExampleCylinder(ThreeDScene):
 def construct(self):
 axes = ThreeDAxes()
 cylinder = Cylinder(radius=2, height=3)
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 self.add(axes, cylinder)
 """

 def __init__(
 self,
 radius: float = 1,
 height: float = 2,

```

```

direction: np.ndarray = Z_AXIS,
v_range: Sequence[float] = [0, TAU],
show_ends: bool = True,
resolution: Sequence[int] = (24, 24),
**kwargs,
) -> None:
 self._height = height
 self.radius = radius
 super().__init__(
 self.func,
 resolution=resolution,
 u_range=[-self._height / 2, self._height / 2],
 v_range=v_range,
 **kwargs,
)
 if show_ends:
 self.add_bases()
 self._current_phi = 0
 self._current_theta = 0
 self.set_direction(direction)

def func(self, u: float, v: float) -> np.ndarray:
 """Converts from cylindrical coordinates to cartesian.

Parameters

u
 The height.
v
 The azimuthal angle.

Returns

:class:`numpy.ndarray`
 Points defining the :class:`Cylinder`.
"""
 height = u
 phi = v
 r = self.radius
 return np.array([r * np.cos(phi), r * np.sin(phi), height])

def add_bases(self) -> None:
 """Adds the end caps of the cylinder."""
 if config.renderer == RendererType.OPENGL:
 color = self.color
 opacity = self.opacity
 elif config.renderer == RendererType.CAIRO:
 color = self.fill_color
 opacity = self.fill_opacity

 self.base_top = Circle(
 radius=self.radius,
 color=color,
 fill_opacity=opacity,
 shade_in_3d=True,
 stroke_width=0,
)
 self.base_top.shift(self.u_range[1] * IN)
 self.base_bottom = Circle(
 radius=self.radius,
 color=color,
 fill_opacity=opacity,
 shade_in_3d=True,
 stroke_width=0,
)
 self.base_bottom.shift(self.u_range[0] * IN)
 self.add(self.base_top, self.base_bottom)

def _rotate_to_direction(self):
 x, y, z = self.direction

 r = np.sqrt(x**2 + y**2 + z**2)
 if r > 0:
 theta = np.arccos(z / r)
 else:
 theta = 0

 if x == 0:
 if y == 0: # along the z axis
 phi = 0
 else: # along the x axis
 phi = np.arctan(np.inf)
 if y < 0:
 phi += PI
 else:
 phi = np.arctan(y / x)
 if x < 0:
 phi += PI

 # undo old rotation (in reverse direction)
 self.rotate(-self._current_phi, Z_AXIS, about_point=ORIGIN)
 self.rotate(-self._current_theta, Y_AXIS, about_point=ORIGIN)

 # do new rotation

```

```

 self.rotate(theta, Y_AXIS, about_point=ORIGIN)
 self.rotate(phi, Z_AXIS, about_point=ORIGIN)

 # store new values
 self._current_theta = theta
 self._current_phi = phi

def set_direction(self, direction: np.ndarray) -> None:
 """Sets the direction of the central axis of the :class:`Cylinder`."""

 Parameters

 direction : :class:`numpy.array`
 The direction of the central axis of the :class:`Cylinder`.
 """
 # if get_norm(direction) is get_norm(self.direction):
 # pass
 self.direction = direction
 self._rotate_to_direction()

def get_direction(self) -> np.ndarray:
 """Returns the direction of the central axis of the :class:`Cylinder`."""

 Returns

 direction : :class:`numpy.array`
 The direction of the central axis of the :class:`Cylinder`.
 """
 return self.direction

class Line3D(Cylinder):
 """A cylindrical line, for use in ThreeDScene.

 Parameters

 start
 The start point of the line.
 end
 The end point of the line.
 thickness
 The thickness of the line.
 color
 The color of the line.

 Examples

 .. manim:: ExampleLine3D
 :save_last_frame:

 class ExampleLine3D(ThreeDScene):
 def construct(self):
 axes = ThreeDaxes()
 line = Line3D(start=np.array([0, 0, 0]), end=np.array([2, 2, 2]))
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 self.add(axes, line)
 """

 def __init__(
 self,
 start: np.ndarray = LEFT,
 end: np.ndarray = RIGHT,
 thickness: float = 0.02,
 color: Color = None,
 **kwargs,
):
 self.thickness = thickness
 self.set_start_and_end_attrs(start, end, **kwargs)
 if color is not None:
 self.set_color(color)

 def set_start_and_end_attrs(
 self, start: np.ndarray, end: np.ndarray, **kwargs
) -> None:
 """Sets the start and end points of the line.

 If either ``start`` or ``end`` are :class:`Mobjects <.Mobject>`, this gives their centers.

 Parameters

 start
 Starting point or :class:`Mobject`.
 end
 Ending point or :class:`Mobject`.
 """
 rough_start = self.pointify(start)
 rough_end = self.pointify(end)
 self.vect = rough_end - rough_start
 self.length = np.linalg.norm(self.vect)
 self.direction = normalize(self.vect)
 # Now that we know the direction between them, we can get the appropriate boundary point from start and end, if they're mobjects

```

```

self.start = self.pointify(start, self.direction)
self.end = self.pointify(end, -self.direction)
super().__init__(
 height=np.linalg.norm(self.vect),
 radius=self.thickness,
 direction=self.direction,
 **kwargs,
)
self.shift((self.start + self.end) / 2)

def pointify(
 self, mob_or_point: Mobject | float, direction: np.ndarray = None
) -> np.ndarray:
 """Gets a point representing the center of the :class:`Mobjects <.Mobject>`."""

 Parameters

 mob_or_point
 :class:`Mobjects <.Mobject>` or point whose center should be returned.
 direction
 If an edge of a :class:`Mobjects <.Mobject>` should be returned, the direction of the edge.

 Returns

 :class:`numpy.array`
 Center of the :class:`Mobjects <.Mobject>` or point, or edge if direction is given.
 """
 if isinstance(mob_or_point, (Mobject, OpenGLMobject)):
 mob = mob_or_point
 if direction is None:
 return mob.get_center()
 else:
 return mob.get_boundary_point(direction)
 return np.array(mob_or_point)

def get_start(self) -> np.ndarray:
 """Returns the starting point of the :class:`Line3D`."""

 Returns

 start : :class:`numpy.array`
 Starting point of the :class:`Line3D`.
 """
 return self.start

def get_end(self) -> np.ndarray:
 """Returns the ending point of the :class:`Line3D`."""

 Returns

 end : :class:`numpy.array`
 Ending point of the :class:`Line3D`.
 """
 return self.end

@classmethod
def parallel_to(
 cls, line: Line3D, point: Sequence[float] = ORIGIN, length: float = 5, **kwargs
) -> Line3D:
 """Returns a line parallel to another line going through
 a given point.

 Parameters

 line
 The line to be parallel to.
 point
 The point to pass through.
 length
 Length of the parallel line.
 kwargs
 Additional parameters to be passed to the class.

 Returns

 :class:`Line3D`
 Line parallel to ``line``.

 Examples

 .. manim:: ParallelLineExample
 :save_last_frame:

 class ParallelLineExample(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(PI / 3, -PI / 4)
 ax = ThreeDAxes((-5, 5), (-5, 5), (-5, 5), 10, 10, 10)
 line1 = Line3D(RIGHT * 2, UP + OUT, color=RED)
 line2 = Line3D.parallel_to(line1, color=YELLOW)
 self.add(ax, line1, line2)

 """
 point = np.array(point)
 vect = normalize(line.vect)
 return cls(

```

```

 point + vect * length / 2,
 point - vect * length / 2,
 **kwargs,
)

@classmethod
def perpendicular_to(
 cls, line: Line3D, point: Sequence[float] = ORIGIN, length: float = 5, **kwargs
) -> Line3D:
 """Returns a line perpendicular to another line going through
 a given point.

 Parameters

 line
 The line to be perpendicular to.
 point
 The point to pass through.
 length
 Length of the perpendicular line.
 kwargs
 Additional parameters to be passed to the class.

 Returns

 :class:`Line3D`
 Line perpendicular to ``line``.

 Examples

 .. manim:: PerpLineExample
 :save_last_frame:

 class PerpLineExample(ThreeDScene):
 def construct(self):
 self.set_camera_orientation(PI / 3, -PI / 4)
 ax = ThreeDAxes((-5, 5), (-5, 5), (-5, 5), 10, 10, 10)
 line1 = Line3D(RIGHT * 2, UP + OUT, color=RED)
 line2 = Line3D.perpendicular_to(line1, color=BLUE)
 self.add(ax, line1, line2)

 """
 point = np.array(point)

 norm = np.cross(line.vect, point - line.start)
 if all(np.linalg.norm(norm) == np.zeros(3)):
 raise ValueError("Could not find the perpendicular.")

 start, end = perpendicular_bisector([line.start, line.end], norm)
 vect = normalize(end - start)
 return cls(
 point + vect * length / 2,
 point - vect * length / 2,
 **kwargs,
)

 class Arrow3D(Line3D):
 """An arrow made out of a cylindrical line and a conical tip.

 Parameters

 start
 The start position of the arrow.
 end
 The end position of the arrow.
 thickness
 The thickness of the arrow.
 height
 The height of the conical tip.
 base_radius
 The base radius of the conical tip.
 color
 The color of the arrow.

 Examples

 .. manim:: ExampleArrow3D
 :save_last_frame:

 class ExampleArrow3D(ThreeDScene):
 def construct(self):
 axes = ThreeDAxes()
 arrow = Arrow3D(
 start=np.array([0, 0, 0]),
 end=np.array([2, 2, 2]),
 resolution=8
)
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 self.add(axes, arrow)

 """

 def __init__(self,
 start: np.ndarray = LEFT,

```

```

end: np.ndarray = RIGHT,
thickness: float = 0.02,
height: float = 0.3,
base_radius: float = 0.08,
color: Color = WHITE,
**kwargs,
) -> None:
 super().__init__(
 start=start, end=end, thickness=thickness, color=color, **kwargs
)

 self.length = np.linalg.norm(self.vect)
 self.set_start_and_end_attrs(
 self.start,
 self.end - height * self.direction,
 **kwargs,
)

 self.cone = Cone(
 direction=self.direction, base_radius=base_radius, height=height, **kwargs
)
 self.cone.shift(end)
 self.add(self.cone)
 self.set_color(color)

class Torus(Surface):
 """A torus.

 Parameters

 major_radius
 Distance from the center of the tube to the center of the torus.
 minor_radius
 Radius of the tube.
 u_range
 The range of the ``u`` variable: ``(``u_min, u_max````.
 v_range
 The range of the ``v`` variable: ``(``v_min, v_max````.
 resolution
 The number of samples taken of the :class:`Torus`. A tuple can be
 used to define different resolutions for ``u`` and ``v`` respectively.

 Examples

 .. manim :: ExampleTorus
 :save_last_frame:

 class ExampleTorus(ThreeDScene):
 def construct(self):
 axes = ThreeDAxes()
 torus = Torus()
 self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
 self.add(axes, torus)
 """

 def __init__(self,
 major_radius: float = 3,
 minor_radius: float = 1,
 u_range: Sequence[float] = (0, TAU),
 v_range: Sequence[float] = (0, TAU),
 resolution: Sequence[int] = None,
 **kwargs,
) -> None:
 if config.renderer == RendererType.OPENGL:
 res_value = (101, 101)
 elif config.renderer == RendererType.CAIRO:
 res_value = (24, 24)

 resolution = resolution if resolution is not None else res_value

 self.R = major_radius
 self.r = minor_radius
 super().__init__(
 self.func,
 u_range=u_range,
 v_range=v_range,
 resolution=resolution,
 **kwargs,
)

 def func(self, u: float, v: float) -> np.ndarray:
 """The z values defining the :class:`Torus` being plotted.

 Returns

 :class:`numpy.ndarray`
 The z values defining the :class:`Torus`.
 """
 P = np.array([np.cos(u), np.sin(u), 0])
 return (self.R - self.r * np.cos(v)) * P - self.r * np.sin(v) * OUT

```

## opengl\_vectorized\_mobject.py

```
from __future__ import annotations

import itertools as it
import operator as op
from functools import reduce, wraps
from typing import Callable, Iterable, Optional, Sequence

import moderngl
import numpy as np
from colour import Color

from manim import config
from manim.constants import *
from manim.mobject.opengl.opengl_mobject import OpenGLMObject, OpenGLPoint
from manim.renderer.shader_wrapper import ShaderWrapper
from manim.utils.bezier import (
 bezier,
 get_quadratic_approximation_of_cubic,
 get_smooth_cubic_bezier_handle_points,
 integer_interpolate,
 interpolate,
 partial_quadratic_bezier_points,
 proportions_along_bezier_curve_for_point,
 quadratic_bezier_remap,
)
from manim.utils.color import *
from manim.utils.config_ops import _Data
from manim.utils.iterables import listify, make_even, resize_with_interpolation
from manim.utils.space_ops import (
 angle_between_vectors,
 cross2d,
 earclip_triangulation,
 get_unit_normal,
 shoelace_direction,
 z_to_vector,
)
)

def triggers_refreshed_triangulation(func):
 @wraps(func)
 def wrapper(self, *args, **kwargs):
 old_points = np.empty((0, 3))
 for mob in self.family_members_with_points():
 old_points = np.concatenate((old_points, mob.points), axis=0)
 func(self, *args, **kwargs)
 new_points = np.empty((0, 3))
 for mob in self.family_members_with_points():
 new_points = np.concatenate((new_points, mob.points), axis=0)
 if not np.array_equal(new_points, old_points):
 self.refresh_triangulation()
 self.refresh_unit_normal()
 return self

 return wrapper

class OpenGLMObject(OpenGLMObject):
 """A vectorized mobject."""

 fill_dtype = [
 ("point", np.float32, (3,)),
 ("unit_normal", np.float32, (3,)),
 ("color", np.float32, (4,)),
 ("vert_index", np.float32, (1,)),
]
 stroke_dtype = [
 ("point", np.float32, (3,)),
 ("prev_point", np.float32, (3,)),
 ("next_point", np.float32, (3,)),
 ("unit_normal", np.float32, (3,)),
 ("stroke_width", np.float32, (1,)),
 ("color", np.float32, (4,)),
]
 stroke_shader_folder = "quadratic_bezier_stroke"
 fill_shader_folder = "quadratic_bezier_fill"

 fill_rgba = _Data()
 stroke_rgba = _Data()
 stroke_width = _Data()
 unit_normal = _Data()

 def __init__(
 self,
 fill_color: Color | None = None,
 fill_opacity: float = 0.0,
 stroke_color: Color | None = None,
 stroke_opacity: float = 1.0,
 stroke_width: float = DEFAULT_STROKE_WIDTH,
 draw_stroke_behind_fill: bool = False,
 # Indicates that it will not be displayed, but
 # that it should count in parent mobject's path
 pre_function_handle_to_anchor_scale_factor: float = 0.01,
 make_smooth_after_applying_functions: float = False,
 background_image_file: str | None = None,
 # This is within a pixel
 # TODO, do we care about accounting for
 # varying zoom levels?
):
```

```

tolerance_for_point_equality: float = 1e-8,
n_points_per_curve: int = 3,
long_lines: bool = False,
should_subdivide_sharp_curves: bool = False,
should_remove_null_curves: bool = False,
Could also be "bevel", "miter", "round"
joint_type: LineJointType | None = None,
flat_stroke: bool = True,
render_primitive=moderngl.TRIANGLES,
triangulation_locked: bool = False,
**kwargs,
):
 self.data = {}
 self.fill_opacity = fill_opacity
 self.stroke_opacity = stroke_opacity
 self.stroke_width = stroke_width
 self.draw_stroke_behind_fill = draw_stroke_behind_fill
 # Indicates that it will not be displayed, but
 # that it should count in parent mobject's path
 self.pre_function_handle_to_anchor_scale_factor = (
 pre_function_handle_to_anchor_scale_factor
)
 self.make_smooth_after_applying_functions = make_smooth_after_applying_functions
 self.background_image_file = background_image_file
 # This is within a pixel
 # TODO, do we care about accounting for
 # varying zoom levels?
 self.tolerance_for_point_equality = tolerance_for_point_equality
 self.n_points_per_curve = n_points_per_curve
 self.long_lines = long_lines
 self.should_subdivide_sharp_curves = should_subdivide_sharp_curves
 self.should_remove_null_curves = should_remove_null_curves
 if joint_type is None:
 joint_type = LineJointType.AUTO
 self.joint_type = joint_type
 self.flat_stroke = flat_stroke
 self.render_primitive = render_primitive
 self.triangulation_locked = triangulation_locked

 self.needs_new_triangulation = True
 self.triangulation = np.zeros(0, dtype="i4")
 self.orientation = 1
 self.fill_data = None
 self.stroke_data = None
 self.fill_shader_wrapper = None
 self.stroke_shader_wrapper = None
 self.init_shader_data()

super().__init__(**kwargs)
self.refresh_unit_normal()

if fill_color:
 self.fill_color = Color(fill_color)
if stroke_color:
 self.stroke_color = Color(stroke_color)

def get_group_class(self):
 return OpenGLGroup

@staticmethod
def get_mobject_type_class():
 return OpenGLMobject

def init_data(self):
 super().init_data()
 self.data.pop("rgbas")
 self.fill_rgba = np.zeros((1, 4))
 self.stroke_rgba = np.zeros((1, 4))
 self.unit_normal = np.zeros((1, 3))
 # stroke_width belongs to self.data, but is defined through init_colors+set_stroke

 # Colors
 def init_colors(self):
 self.set_fill(
 color=self.fill_color or self.color,
 opacity=self.fill_opacity,
)
 self.set_stroke(
 color=self.stroke_color or self.color,
 width=self.stroke_width,
 opacity=self.stroke_opacity,
 background=self.draw_stroke_behind_fill,
)
 self.set_gloss(self.gloss)
 self.set_flat_stroke(self.flat_stroke)
 return self

def set_fill(
 self,
 color: Color | None = None,
 opacity: float | None = None,
 recurse: bool = True,
) -> OpenGLMobject:
 """Set the fill color and fill opacity of a :class:`OpenGLMobject`."""

 Parameters

 color
 Fill color of the :class:`OpenGLMobject`.
 opacity
 Fill opacity of the :class:`OpenGLMobject`.
 recurse
 If ``True``, the fill color of all subobjects is also set.

```

```

>Returns

OpenGLVMOBJECT
 self. For chaining purposes.

.. manim:: SetFill
 :save_last_frame:

 class SetFill(Scene):
 def construct(self):
 square = Square().scale(2).set_fill(WHITE,1)
 circle1 = Circle().set_fill(GREEN,0.8)
 circle2 = Circle().set_fill(YELLOW) # No fill_opacity
 circle3 = Circle().set_fill(color = '#FF2135', opacity = 0.2)
 group = Group(circle1,circle2,circle3).arrange()
 self.add(square)
 self.add(group)

See Also

:meth:`~.OpenGLVMOBJECT.set_style`

if opacity is not None:
 self.fill_opacity = opacity
if recurse:
 for submob in self.submobjects:
 submob.set_fill(color, opacity, recurse)

self.set_rgba_array(color, opacity, "fill_rgba", recurse)
return self

def set_stroke(
 self,
 color=None,
 width=None,
 opacity=None,
 background=None,
 recurse=True,
):
 if opacity is not None:
 self.stroke_opacity = opacity
 if recurse:
 for submob in self.submobjects:
 submob.set_stroke(
 color=color,
 width=width,
 opacity=opacity,
 background=background,
 recurse=recurse,
)
 self.set_rgba_array(color, opacity, "stroke_rgba", recurse)

 if width is not None:
 for mob in self.get_family(recurse):
 mob.stroke_width = np.array([[width] for width in listify(width)])

 if background is not None:
 for mob in self.get_family(recurse):
 mob.draw_stroke_behind_fill = background
 return self

def set_style(
 self,
 fill_color=None,
 fill_opacity=None,
 fill_rgba=None,
 stroke_color=None,
 stroke_opacity=None,
 stroke_rgba=None,
 stroke_width=None,
 gloss=None,
 shadow=None,
 recurse=True,
):
 if fill_rgba is not None:
 self.fill_rgba = resize_with_interpolation(fill_rgba, len(fill_rgba))
 else:
 self.set_fill(color=fill_color, opacity=fill_opacity, recurse=recurse)

 if stroke_rgba is not None:
 self.stroke_rgba = resize_with_interpolation(stroke_rgba, len(fill_rgba))
 self.set_stroke(width=stroke_width)
 else:
 self.set_stroke(
 color=stroke_color,
 width=stroke_width,
 opacity=stroke_opacity,
 recurse=recurse,
)

 if gloss is not None:
 self.set_gloss(gloss, recurse=recurse)
 if shadow is not None:
 self.set_shadow(shadow, recurse=recurse)
 return self

def get_style(self):
 return {
 "fill_rgba": self.fill_rgba,

```

```

 "stroke_rgba": self.stroke_rgba,
 "stroke_width": self.stroke_width,
 "gloss": self.gloss,
 "shadow": self.shadow,
 }

def match_style(self, vmobject, recurse=True):
 vmobject_style = vmobject.get_style()
 if config.renderer == RendererType.OPENGL:
 vmobject_style["stroke_width"] = vmobject_style["stroke_width"][0][0]
 self.set_style(**vmobject_style, recurse=False)
 if recurse:
 # Does its best to match up submobject lists, and
 # match styles accordingly
 submobs1, submobs2 = self.submobjects, vmobject.submobjects
 if len(submobs1) == 0:
 return self
 elif len(submobs2) == 0:
 submobs2 = [vmobject]
 for sm1, sm2 in zip(*make_even(submobs1, submobs2)):
 sm1.match_style(sm2)
 return self

def set_color(self, color, opacity=None, recurse=True):
 if opacity is not None:
 self.opacity = opacity

 self.set_fill(color, opacity=opacity, recurse=recurse)
 self.set_stroke(color, opacity=opacity, recurse=recurse)
 return self

def set_opacity(self, opacity, recurse=True):
 self.set_fill(opacity=opacity, recurse=recurse)
 self.set_stroke(opacity=opacity, recurse=recurse)
 return self

def fade(self, darkness=0.5, recurse=True):
 factor = 1.0 - darkness
 self.set_fill(
 opacity=factor * self.get_fill_opacity(),
 recurse=False,
)
 self.set_stroke(
 opacity=factor * self.get_stroke_opacity(),
 recurse=False,
)
 super().fade(darkness, recurse)
 return self

def get_fill_colors(self):
 return [Color(rgb_to_hex(rgba[:3])) for rgba in self.fill_rgba]

def get_fill_opacities(self):
 return self.fill_rgba[:, 3]

def get_stroke_colors(self):
 return [Color(rgb_to_hex(rgba[:3])) for rgba in self.stroke_rgba]

def get_stroke_opacities(self):
 return self.stroke_rgba[:, 3]

def get_stroke_widths(self):
 return self.stroke_width

TODO, it's weird for these to return the first of various lists
rather than the full information
def get_fill_color(self):
 """
 If there are multiple colors (for gradient)
 this returns the first one
 """
 return self.get_fill_colors()[0]

def get_fill_opacity(self):
 """
 If there are multiple opacities, this returns the
 first
 """
 return self.get_fill_opacities()[0]

def get_stroke_color(self):
 return self.get_stroke_colors()[0]

def get_stroke_width(self):
 return self.get_stroke_widths()[0]

def get_stroke_opacity(self):
 return self.get_stroke_opacities()[0]

def get_color(self):
 if self.has_stroke():
 return self.get_stroke_color()
 return self.get_fill_color()

def get_colors(self):
 if self.has_stroke():
 return self.get_stroke_colors()
 return self.get_fill_colors()

stroke_color = property(get_stroke_color, set_stroke)
color = property(get_color, set_color)
fill_color = property(get_fill_color, set_fill)

```

```

def has_stroke(self):
 stroke_widths = self.get_stroke_widths()
 stroke_opacities = self.get_stroke_opacities()
 return (
 stroke_widths is not None
 and stroke_opacities is not None
 and any(stroke_widths)
 and any(stroke_opacities)
)

def has_fill(self):
 fill_opacities = self.get_fill_opacities()
 return fill_opacities is not None and any(fill_opacities)

def get_opacity(self):
 if self.has_fill():
 return self.get_fill_opacity()
 return self.get_stroke_opacity()

def set_flat_stroke(self, flat_stroke=True, recurse=True):
 for mob in self.get_family(recurse):
 mob.flat_stroke = flat_stroke
 return self

def get_flat_stroke(self):
 return self.flat_stroke

Points
def set_anchors_and_handles(self, anchors1, handles, anchors2):
 assert len(anchors1) == len(handles) == len(anchors2)
 nppc = self.n_points_per_curve
 new_points = np.zeros((nppc * len(anchors1), self.dim))
 arrays = [anchors1, handles, anchors2]
 for index, array in enumerate(arrays):
 new_points[index::nppc] = array
 self.set_points(new_points)
 return self

def start_new_path(self, point):
 assert self.get_num_points() % self.n_points_per_curve == 0
 self.append_points([point])
 return self

def add_cubic_bezier_curve(self, anchor1, handle1, handle2, anchor2):
 new_points = get_quadratic_approximation_of_cubic(
 anchor1,
 handle1,
 handle2,
 anchor2,
)
 self.append_points(new_points)

def add_cubic_bezier_curve_to(self, handle1, handle2, anchor):
 """
 Add cubic bezier curve to the path.
 """
 self.throw_error_if_no_points()
 quadratic_approx = get_quadratic_approximation_of_cubic(
 self.get_last_point(),
 handle1,
 handle2,
 anchor,
)
 if self.has_new_path_started():
 self.append_points(quadratic_approx[1:])
 else:
 self.append_points(quadratic_approx)

def add_quadratic_bezier_curve_to(self, handle, anchor):
 self.throw_error_if_no_points()
 if self.has_new_path_started():
 self.append_points([handle, anchor])
 else:
 self.append_points([self.get_last_point(), handle, anchor])

def add_line_to(self, point: Sequence[float]) -> OpenGLMobject:
 """Add a straight line from the last point of OpenGLMobject to the given point.

 Parameters

 point
 end of the straight line.
 """
 end = self.points[-1]
 alphas = np.linspace(0, 1, self.n_points_per_curve)
 if self.long_lines:
 halfway = interpolate(end, point, 0.5)
 points = [interpolate(end, halfway, a) for a in alphas] + [
 interpolate(halfway, point, a) for a in alphas
]
 else:
 points = [interpolate(end, point, a) for a in alphas]
 if self.has_new_path_started():
 points = points[1:]
 self.append_points(points)
 return self

def add_smooth_curve_to(self, point):
 if self.has_new_path_started():
 self.add_line_to(point)
 else:
 self.throw_error_if_no_points()

```

```

 new_handle = self.get_reflection_of_last_handle()
 self.add_quadratic_bezier_curve_to(new_handle, point)
 return self

def add_smooth_cubic_curve_to(self, handle, point):
 self.throw_error_if_no_points()
 new_handle = self.get_reflection_of_last_handle()
 self.add_cubic_bezier_curve_to(new_handle, handle, point)

def has_new_path_started(self):
 return self.get_num_points() % self.n_points_per_curve == 1

def get_last_point(self):
 return self.points[-1]

def get_reflection_of_last_handle(self):
 points = self.points
 return 2 * points[-1] - points[-2]

def close_path(self):
 if not self.is_closed():
 self.add_line_to(self.get_subpaths()[-1][0])

def is_closed(self):
 return self.consider_points_equals(self.points[0], self.points[-1])

def subdivide_sharp_curves(self, angle_threshold=30 * DEGREES, recurse=True):
 vmobs = [vm for vm in self.get_family(recurse) if vm.has_points()]
 for vmob in vmobs:
 new_points = []
 for tup in vmob.get_bezier_tuples():
 angle = angle_between_vectors(tup[1] - tup[0], tup[2] - tup[1])
 if angle > angle_threshold:
 n = int(np.ceil(angle / angle_threshold))
 alphas = np.linspace(0, 1, n + 1)
 new_points.extend([
 partial_quadratic_bezier_points(tup, a1, a2)
 for a1, a2 in zip(alphas, alphas[1:])])
 else:
 new_points.append(tup)
 vmob.set_points(np.vstack(new_points))
 return self

def add_points_as_corners(self, points):
 for point in points:
 self.add_line_to(point)
 return points

def set_points_as_corners(self, points: Iterable[float]) -> OpenGLMobject:
 """Given an array of points, set them as corner of the vmobject.

 To achieve that, this algorithm sets handles aligned with the anchors such that the resultant bezier curve will be the segment between the two anchors.

 Parameters

 points
 Array of points that will be set as corners.

 Returns

 OpenGLMobject
 self. For chaining purposes.
 """
 nppc = self.n_points_per_curve
 points = np.array(points)
 self.set_anchors_and_handles(
 *(interpolate(points[:-1], points[1:], a) for a in np.linspace(0, 1, nppc)))
 return self

def set_points_smoothly(self, points, true_smooth=False):
 self.set_points_as_corners(points)
 self.make_smooth()
 return self

def change_anchor_mode(self, mode):
 """Changes the anchor mode of the bezier curves. This will modify the handles.

 There can be only three modes, "jagged", "approx_smooth" and "true_smooth".

 Returns

 OpenGLMobject
 For chaining purposes.
 """
 assert mode in ("jagged", "approx_smooth", "true_smooth")
 nppc = self.n_points_per_curve
 for submob in self.family_members_with_points():
 subpaths = submob.get_subpaths()
 submob.clear_points()
 for subpath in subpaths:
 anchors = np.vstack([subpath[:nppc], subpath[-1:]])
 new_subpath = np.array(subpath)
 if mode == "approx_smooth":
 # TODO: get_smooth_quadratic_bezier_handle_points is not defined
 new_subpath[1::nppc] = get_smooth_quadratic_bezier_handle_points(
 anchors,
)
 elif mode == "true_smooth":

```

```

 h1, h2 = get_smooth_cubic_bezier_handle_points(anchors)
 new_subpath = get_quadratic_approximation_of_cubic(
 anchors[:-1],
 h1,
 h2,
 anchors[1:],
)
 elif mode == "jagged":
 new_subpath[1::nppc] = 0.5 * (anchors[:-1] + anchors[1:])
 submob.append_points(new_subpath)
 submob.refresh_triangulation()
return self

def make_smooth(self):
 """
 This will double the number of points in the mobject,
 so should not be called repeatedly. It also means
 transforming between states before and after calling
 this might have strange artifacts
 """
 self.change_anchor_mode("true_smooth")
 return self

def make_approximately_smooth(self):
 """
 Unlike make_smooth, this will not change the number of
 points, but it also does not result in a perfectly smooth
 curve. It's most useful when the points have been
 sampled at a not-too-low rate from a continuous function,
 as in the case of ParametricCurve
 """
 self.change_anchor_mode("approx_smooth")
 return self

def make_jagged(self):
 self.change_anchor_mode("jagged")
 return self

def add_subpath(self, points):
 assert len(points) % self.n_points_per_curve == 0
 self.append_points(points)
 return self

def append_vectorized_mobject(self, vectorized_mobject):
 new_points = list(vectorized_mobject.points)

 if self.has_new_path_started():
 # Remove last point, which is starting
 # a new path
 self.resize_data(len(self.points) - 1)
 self.append_points(new_points)
 return self

#
def consider_points_equals(self, p0, p1):
return np.linalg.norm(p1 - p0) < self.tolerance_for_point_equality

Information about the curve
def force_direction(self, target_direction: str):
 """Makes sure that points are either directed clockwise or
 counterclockwise.

 Parameters

 target_direction
 Either ``"CW"`` or ``"CCW"``.
 """
 if target_direction not in ("CW", "CCW"):
 raise ValueError('Invalid input for force_direction. Use "CW" or "CCW"')

 if self.get_direction() != target_direction:
 self.reverse_points()

 return self

def reverse_direction(self):
 """Reverts the point direction by inverting the point order.

 Returns

 :class:`OpenGLVMMobject`
 Returns self.
 """

Examples

.. manim:: ChangeOfDirection

 class ChangeOfDirection(Scene):
 def construct(self):
 ccw = RegularPolygon(5)
 ccw.shift(LEFT)
 cw = RegularPolygon(5)
 cw.shift(RIGHT).reverse_direction()

 self.play(Create(ccw), Create(cw),
 run_time=4)
 """
 self.set_points(self.points[::-1])
 return self

def get_bezier_tuples_from_points(self, points):
 nppc = self.n_points_per_curve
 remainder = len(points) % nppc

```

```

points = points[: len(points) - remainder]
return points.reshape((-1, nppc, 3))

def get_bezier_tuples(self):
 return self.get_bezier_tuples_from_points(self.points)

def get_subpaths_from_points(self, points):
 nppc = self.n_points_per_curve
 diffs = points[nppc - 1 : -1 : nppc] - points[nppc::nppc]
 splits = (diffs * diffs).sum(1) > self.tolerance_for_point_equality
 split_indices = np.arange(nppc, len(points), nppc, dtype=int)[splits]

 # split_indices = filter(
 # lambda n: not self.consider_points_equals(points[n - 1], points[n]),
 # range(nppc, len(points), nppc)
 #)
 split_indices = [0, *split_indices, len(points)]
 return [
 points[i1:i2]
 for i1, i2 in zip(split_indices, split_indices[1:])
 if (i2 - i1) >= nppc
]

def get_subpaths(self):
 """Returns subpaths formed by the curves of the OpenGLMobject.

 Subpaths are ranges of curves with each pair of consecutive
 curves having their end/start points coincident.

 Returns

 Tuple
 subpaths.
 """
 return self.get_subpaths_from_points(self.points)

def get_nth_curve_points(self, n: int) -> np.ndarray:
 """Returns the points defining the nth curve of the vmobject.

 Parameters

 n
 index of the desired bezier curve.

 Returns

 np.ndarray
 points defininf the nth bezier curve (anchors, handles)
 """
 assert n < self.get_num_curves()
 nppc = self.n_points_per_curve
 return self.points[nppc * n : nppc * (n + 1)]

def get_nth_curve_function(self, n: int) -> Callable[[float], np.ndarray]:
 """Returns the expression of the nth curve.

 Parameters

 n
 index of the desired curve.

 Returns

 typing.Callable[float]
 expression of the nth bezier curve.
 """
 return bezier(self.get_nth_curve_points(n))

def get_nth_curve_function_with_length(
 self,
 n: int,
 sample_points: int | None = None,
) -> tuple[Callable[[float], np.ndarray], float]:
 """Returns the expression of the nth curve along with its (approximate) length.

 Parameters

 n
 The index of the desired curve.
 sample_points
 The number of points to sample to find the length.

 Returns

 curve : Callable[[float], np.ndarray]
 The function for the nth curve.
 length : :class:`float`
 The length of the nth curve.
 """
 if sample_points is None:
 sample_points = 10

 curve = self.get_nth_curve_function(n)
 norms = self.get_nth_curve_length_pieces(n, sample_points)

 length = np.sum(norms)

 return curve, length

def get_num_curves(self) -> int:
 """Returns the number of curves of the vmobject.

```

```

 Returns

 int
 number of curves. of the vmobject.
 """
 return self.get_num_points() // self.n_points_per_curve

def get_nth_curve_length(
 self,
 n: int,
 sample_points: int | None = None,
) -> float:
 """Returns the (approximate) length of the nth curve.

 Parameters

 n
 The index of the desired curve.
 sample_points
 The number of points to sample to find the length.

 Returns

 length : :class:`float`
 The length of the nth curve.
 """

 _, length = self.get_nth_curve_function_with_length(n, sample_points)
 return length

def get_curve_functions(
 self,
) -> Iterable[Callable[[float], np.ndarray]]:
 """Gets the functions for the curves of the mobject.

 Returns

 Iterable[Callable[[float], np.ndarray]]
 The functions for the curves.
 """

 num_curves = self.get_num_curves()

 for n in range(num_curves):
 yield self.get_nth_curve_function(n)

def get_nth_curve_length_pieces(
 self,
 n: int,
 sample_points: int | None = None,
) -> np.ndarray:
 """Returns the array of short line lengths used for length approximation.

 Parameters

 n
 The index of the desired curve.
 sample_points
 The number of points to sample to find the length.

 Returns

 np.ndarray
 The short length-pieces of the nth curve.
 """

 if sample_points is None:
 sample_points = 10

 curve = self.get_nth_curve_function(n)
 points = np.array([curve(a) for a in np.linspace(0, 1, sample_points)])
 diffs = points[1:] - points[:-1]
 norms = np.apply_along_axis(np.linalg.norm, 1, diffs)

 return norms

def get_curve_functions_with_lengths(
 self, **kwargs
) -> Iterable[tuple[Callable[[float], np.ndarray], float]]:
 """Gets the functions and lengths of the curves for the mobject.

 Parameters

 **kwargs
 The keyword arguments passed to :meth:`get_nth_curve_function_with_length`

 Returns

 Iterable[Tuple[Callable[[float], np.ndarray], float]]
 The functions and lengths of the curves.
 """

 num_curves = self.get_num_curves()

 for n in range(num_curves):
 yield self.get_nth_curve_function_with_length(n, **kwargs)

def point_from_proportion(self, alpha: float) -> np.ndarray:
 """Gets the point at a proportion along the path of the :class:`OpenGLVMOBJECT` .

 Parameters

 alpha

```

```

The proportion along the the path of the :class:`OpenGLMobject`.

>Returns

:class:`numpy.ndarray`
 The point on the :class:`OpenGLMobject`.

>Raises

:exc:`ValueError`
 If ``alpha`` is not between 0 and 1.
:exc:`Exception`
 If the :class:`OpenGLMobject` has no points.
"""

if alpha < 0 or alpha > 1:
 raise ValueError(f"Alpha {alpha} not between 0 and 1.")

self.throw_error_if_no_points()
if alpha == 1:
 return self.points[-1]

curves_and_lengths = tuple(self.get_curve_functions_with_lengths())
target_length = alpha * np.sum(length for _, length in curves_and_lengths)
current_length = 0

for curve, length in curves_and_lengths:
 if current_length + length >= target_length:
 if length != 0:
 residue = (target_length - current_length) / length
 else:
 residue = 0

 return curve(residue)

 current_length += length

def proportion_from_point(
 self,
 point: Iterable[float | int],
) -> float:
 """Returns the proportion along the path of the :class:`OpenGLMobject` a particular given point is at.

 Parameters

 point
 The Cartesian coordinates of the point which may or may not lie on the :class:`OpenGLMobject`."""

>Returns

float
 The proportion along the path of the :class:`OpenGLMobject`.

>Raises

:exc:`ValueError`
 If ``point`` does not lie on the curve.
:exc:`Exception`
 If the :class:`OpenGLMobject` has no points.
"""
self.throw_error_if_no_points()

Iterate over each bezier curve that the ``VMobject`` is composed of, checking
if the point lies on that curve. If it does not lie on that curve, add
the whole length of the curve to ``target_length`` and move onto the next
curve. If the point does lie on the curve, add how far along the curve
the point is to ``target_length``.
Then, divide ``target_length`` by the total arc length of the shape to get
the proportion along the ``VMobject`` the point is at.

num_curves = self.get_num_curves()
total_length = self.get_arc_length()
target_length = 0
for n in range(num_curves):
 control_points = self.get_nth_curve_points(n)
 length = self.get_nth_curve_length(n)
 proportions_along_bezier = proportions_along_bezier_curve_for_point(
 point,
 control_points,
)
 if len(proportions_along_bezier) > 0:
 proportion_along_nth_curve = max(proportions_along_bezier)
 target_length += length * proportion_along_nth_curve
 break
 target_length += length
else:
 raise ValueError(f"Point {point} does not lie on this curve.")

alpha = target_length / total_length

return alpha

def get_anchors_and_handles(self):
 """
 Returns anchors1, handles, anchors2,
 where (anchors1[i], handles[i], anchors2[i])
 will be three points defining a quadratic bezier curve
 for any i in range(0, len(anchors1))
 """
 nppc = self.n_points_per_curve
 points = self.points

```

```

 return [points[i::nppc] for i in range(nppc)]

def get_start_anchors(self) -> np.ndarray:
 """Returns the start anchors of the bezier curves.

 Returns

 np.ndarray
 Starting anchors
 """
 return self.points[0 :: self.n_points_per_curve]

def get_end_anchors(self) -> np.ndarray:
 """Return the starting anchors of the bezier curves.

 Returns

 np.ndarray
 Starting anchors
 """
 nppc = self.n_points_per_curve
 return self.points[nppc - 1 :: nppc]

def get_anchors(self) -> np.ndarray:
 """Returns the anchors of the curves forming the OpenGLMobject.

 Returns

 np.ndarray
 The anchors.
 """
 points = self.points
 if len(points) == 1:
 return points
 return np.array(
 list(
 it.chain(
 *zip(
 self.get_start_anchors(),
 self.get_end_anchors(),
)
),
),
)

def get_points_without_null_curves(self, atol=1e-9):
 nppc = self.n_points_per_curve
 points = self.points
 distinct_curves = reduce(
 op.or_,
 [
 (abs(points[i::nppc] - points[0::nppc]) > atol).any(1)
 for i in range(1, nppc)
],
)
 return points[distinct_curves.repeat(nppc)]

def get_arc_length(self, sample_points_per_curve: int | None = None) -> float:
 """Return the approximated length of the whole curve.

 Parameters

 sample_points_per_curve
 Number of sample points per curve used to approximate the length. More points result in a better approximation.

 Returns

 float
 The length of the :class:`OpenGLMobject`.
 """

 return np.sum(
 length
 for _, length in self.get_curve_functions_with_lengths(
 sample_points=sample_points_per_curve,
)
)

def get_area_vector(self):
 # Returns a vector whose length is the area bound by
 # the polygon formed by the anchor points, pointing
 # in a direction perpendicular to the polygon according
 # to the right hand rule.
 if not self.has_points():
 return np.zeros(3)

 nppc = self.n_points_per_curve
 points = self.points
 p0 = points[0::nppc]
 p1 = points[nppc - 1 :: nppc]

 # Each term goes through all edges [(x1, y1, z1), (x2, y2, z2)]
 return 0.5 * np.array(
 [
 sum(
 (p0[:, 1] + p1[:, 1]) * (p1[:, 2] - p0[:, 2]),
), # Add up (y1 + y2)*(z2 - z1)
 sum(
 (p0[:, 2] + p1[:, 2]) * (p1[:, 0] - p0[:, 0]),
), # Add up (z1 + z2)*(x2 - x1)
 sum(
 (p0[:, 0] + p1[:, 0]) * (p1[:, 1] - p0[:, 1]),
), # Add up (x1 + x2)*(y2 - y1)
]
)

```

```

],
)

def get_direction(self):
 """Uses :func:`~.space_ops.shoelace_direction` to calculate the direction.
 The direction of points determines in which direction the
 object is drawn, clockwise or counterclockwise.

 Examples

 The default direction of a :class:`~.Circle` is counterclockwise::

 >>> from manim import Circle
 >>> Circle().get_direction()
 'CCW'

 Returns

 :class:`str`
 Either ``"CW"`` or ``"CCW"``.

 """
 return shoelace_direction(self.get_start_anchors())

def get_unit_normal(self, recompute=False):
 if not recompute:
 return self.unit_normal[0]

 if len(self.points) < 3:
 return OUT

 area_vect = self.get_area_vector()
 area = np.linalg.norm(area_vect)
 if area > 0:
 return area_vect / area
 else:
 points = self.points
 return get_unit_normal(
 points[1] - points[0],
 points[2] - points[1],
)

def refresh_unit_normal(self):
 for mob in self.get_family():
 mob.unit_normal[:] = mob.get_unit_normal(recompute=True)
 return self

Alignment
def align_points(self, vmobject):
 # TODO: This shortcut can be a bit over eager. What if they have the same length, but different subpath lengths?
 if self.get_num_points() == len(vmobject.points):
 return

 for mob in self, vmobject:
 # If there are no points, add one to
 # where the "center" is
 if not mob.has_points():
 mob.start_new_path(mob.get_center())
 # If there's only one point, turn it into
 # a null curve
 if mob.has_new_path_started():
 mob.add_line_to(mob.points[0])

 # Figure out what the subpaths are, and align
 subpaths1 = self.get_subpaths()
 subpaths2 = vmobject.get_subpaths()
 n_subpaths = max(len(subpaths1), len(subpaths2))
 # Start building new ones
 new_subpaths1 = []
 new_subpaths2 = []

 nppc = self.n_points_per_curve

 def get_nth_subpath(path_list, n):
 if n >= len(path_list):
 # Create a null path at the very end
 return [path_list[-1][-1]] * nppc
 path = path_list[n]
 # Check for useless points at the end of the path and remove them
 # https://github.com/ManimCommunity/manim/issues/1959
 while len(path) > nppc:
 # If the last nppc points are all equal to the preceding point
 if self.consider_points_equals(path[-nppc:], path[-nppc - 1]):
 path = path[:-nppc]
 else:
 break
 return path

 for n in range(n_subpaths):
 sp1 = get_nth_subpath(subpaths1, n)
 sp2 = get_nth_subpath(subpaths2, n)
 diff1 = max(0, (len(sp2) - len(sp1)) // nppc)
 diff2 = max(0, (len(sp1) - len(sp2)) // nppc)
 sp1 = self.insert_n_curves_to_point_list(diff1, sp1)
 sp2 = self.insert_n_curves_to_point_list(diff2, sp2)
 new_subpaths1.append(sp1)
 new_subpaths2.append(sp2)
 self.set_points(np.vstack(new_subpaths1))
 vmobject.set_points(np.vstack(new_subpaths2))
 return self

def insert_n_curves(self, n: int, recurse=True) -> OpenGLVMobject:
 """Inserts n curves to the bezier curves of the vmobject.

```

```

Parameters

n
 Number of curves to insert.

Returns

OpenGLVMobject
 for chaining.
"""
for mob in self.get_family(recurse):
 if mob.get_num_curves() > 0:
 new_points = mob.insert_n_curves_to_point_list(n, mob.points)
 # TODO, this should happen in insert_n_curves_to_point_list
 if mob.has_new_path_started():
 new_points = np.vstack([new_points, mob.get_last_point()])
 mob.set_points(new_points)
return self

def insert_n_curves_to_point_list(self, n: int, points: np.ndarray) -> np.ndarray:
 """Given an array of k points defining a bezier curves
 (anchors and handles), returns points defining exactly
 k + n bezier curves.

Parameters

n
 Number of desired curves.
points
 Starting points.

Returns

np.ndarray
 Points generated.
"""
nppc = self.n_points_per_curve
if len(points) == 1:
 return np.repeat(points, nppc * n, 0)

bezier_groups = self.get_bezier_tuples_from_points(points)
norms = np.array([np.linalg.norm(bg[nppc - 1] - bg[0]) for bg in bezier_groups])
total_norm = sum(norms)
Calculate insertions per curve (ipc)
if total_norm < 1e-6:
 ipc = [n] + [0] * (len(bezier_groups) - 1)
else:
 ipc = np.round(n * norms / sum(norms)).astype(int)

diff = n - sum(ipc)
for _ in range(diff):
 ipc[np.argmin(ipc)] += 1
for _ in range(-diff):
 ipc[np.argmax(ipc)] -= 1

new_points = []
for group, n_inserts in zip(bezier_groups, ipc):
 # What was once a single quadratic curve defined
 # by "group" will now be broken into n_inserts + 1
 # smaller quadratic curves
 alphas = np.linspace(0, 1, n_inserts + 2)
 for a1, a2 in zip(alphas, alphas[1:]):
 new_points += partial_quadratic_bezier_points(group, a1, a2)
return np.vstack(new_points)

def interpolate(self, mobject1, mobject2, alpha, *args, **kwargs):
 super().interpolate(mobject1, mobject2, alpha, *args, **kwargs)
 if config["use_projection_fill_shaders"]:
 self.refresh_triangulation()
 else:
 if self.has_fill():
 tri1 = mobject1.get_triangulation()
 tri2 = mobject2.get_triangulation()
 if len(tri1) != len(tri2) or not np.all(tri1 == tri2):
 self.refresh_triangulation()
 return self

def pointwiseBecomePartial(
 self, vmobject: OpenGLVMobject, a: float, b: float, remap: bool = True
) -> OpenGLVMobject:
 """Given two bounds a and b, transforms the points of the self vmobject into the points of the vmobject
 passed as parameter with respect to the bounds. Points here stand for control points of the bezier curves (anchors and handles)

Parameters

vmobject
 The vmobject that will serve as a model.
a
 upper-bound.
b
 lower-bound
remap
 if the point amount should be kept the same (True)
 This option should be manually set to False if keeping the number of points is not needed
"""
assert isinstance(vmobject, OpenGLVMobject)
Partial curve includes three portions:
- A middle section, which matches the curve exactly
- A start, which is some ending portion of an inner cubic
- An end, which is the starting portion of a later inner cubic
if a <= 0 and b >= 1:
 self.set_points(vmobject.points)
return self

```

```

bezier_triplets = vmobject.get_bezier_tuples()
num_quadratics = len(bezier_triplets)

The following two lines will compute which bezier curves of the given mobject need to be processed.
The residue basically indicates the proportion of the selected Bézier curve.
Ex: if lower_index is 3, and lower_residue is 0.4, then the algorithm will append to the points 0.4 of the third bezier curve
lower_index, lower_residue = integer_interpolate(0, num_quadratics, a)
upper_index, upper_residue = integer_interpolate(0, num_quadratics, b)
self.clear_points()
if num_quadratics == 0:
 return self
if lower_index == upper_index:
 self.append_points(
 partial_quadratic_bezier_points(
 bezier_triplets[lower_index],
 lower_residue,
 upper_residue,
),
)
else:
 self.append_points(
 partial_quadratic_bezier_points(
 bezier_triplets[lower_index], lower_residue, 1
),
)
inner_points = bezier_triplets[lower_index + 1 : upper_index]
if len(inner_points) > 0:
 if remap:
 new_triplets = quadratic_bezier_remap(
 inner_points, num_quadratics - 2
)
 else:
 new_triplets = bezier_triplets

 self.append_points(np.asarray(new_triplets).reshape(-1, 3))
 self.append_points(
 partial_quadratic_bezier_points(
 bezier_triplets[upper_index], 0, upper_residue
),
)
return self

def get_subcurve(self, a: float, b: float) -> OpenGLMobject:
 """Returns the subcurve of the OpenGLMobject between the interval [a, b].
 The curve is a OpenGLMobject itself.

 Parameters

 a
 The lower bound.
 b
 The upper bound.

 Returns

 OpenGLMobject
 The subcurve between of [a, b]
 """
 vmob = self.copy()
 vmob.pointwise_become_partial(self, a, b)
 return vmob

Related to triangulation

def refresh_triangulation(self):
 for mob in self.get_family():
 mob.needs_new_triangulation = True
 return self

def get_triangulation(self, normal_vector=None):
 # Figure out how to triangulate the interior to know
 # how to send the points as to the vertex shader.
 # First triangles come directly from the points
 if normal_vector is None:
 normal_vector = self.get_unit_normal()

 if not self.needs_new_triangulation:
 return self.triangulation

 points = self.points

 if len(points) <= 1:
 self.triangulation = np.zeros(0, dtype="i4")
 self.needs_new_triangulation = False
 return self.triangulation

 if not np.isclose(normal_vector, OUT).all():
 # Rotate points such that unit normal vector is OUT
 points = np.dot(points, z_to_vector(normal_vector))
 indices = np.arange(len(points), dtype=int)

 b0s = points[0::3]
 b1s = points[1::3]
 b2s = points[2::3]
 v01s = b1s - b0s
 v12s = b2s - b1s

 crosses = cross2d(v01s, v12s)
 convexities = np.sign(crosses)

 atol = self.tolerance_for_point_equality
 end_of_loop = np.zeros(len(b0s), dtype=bool)

```

```

end_of_loop[:-1] = (np.abs(b2s[:-1] - b0s[1:]) > atol).any(1)
end_of_loop[-1] = True

concave_parts = convexities < 0

These are the vertices to which we'll apply a polygon triangulation
inner_vert_indices = np.hstack(
 [
 indices[0::3],
 indices[1::3][concave_parts],
 indices[2::3][end_of_loop],
],
)
inner_vert_indices.sort()
rings = np.arange(1, len(inner_vert_indices) + 1)[inner_vert_indices % 3 == 2]

Triangulate
inner_verts = points[inner_vert_indices]
inner_tri_indices = inner_vert_indices[
 earclip_triangulation(inner_verts, rings)
]

tri_indices = np.hstack([indices, inner_tri_indices])
self.triangulation = tri_indices
self.needs_new_triangulation = False
return tri_indices

@triggers_refreshed_triangulation
def set_points(self, points):
 super().set_points(points)
 return self

@triggers_refreshed_triangulation
def set_data(self, data):
 super().set_data(data)
 return self

TODO, how to be smart about tangents here?
@triggers_refreshed_triangulation
def apply_function(self, function, make_smooth=False, **kwargs):
 super().apply_function(function, **kwargs)
 if self.make_smooth_after_applying_functions or make_smooth:
 self.make_approximately_smooth()
 return self

@triggers_refreshed_triangulation
def apply_points_function(self, *args, **kwargs):
 super().apply_points_function(*args, **kwargs)
 return self

@triggers_refreshed_triangulation
def flip(self, *args, **kwargs):
 super().flip(*args, **kwargs)
 return self

For shaders
def init_shader_data(self):
 self.fill_data = np.zeros(0, dtype=self.fill_dtype)
 self.stroke_data = np.zeros(0, dtype=self.stroke_dtype)
 self.fill_shader_wrapper = ShaderWrapper(
 vert_data=self.fill_data,
 vert_indices=np.zeros(0, dtype="i4"),
 shader_folder=self.fill_shader_folder,
 render_primitive=self.render_primitive,
)
 self.stroke_shader_wrapper = ShaderWrapper(
 vert_data=self.stroke_data,
 shader_folder=self.stroke_shader_folder,
 render_primitive=self.render_primitive,
)

def refresh_shader_wrapper_id(self):
 for wrapper in [self.fill_shader_wrapper, self.stroke_shader_wrapper]:
 wrapper.refresh_id()
 return self

def get_fill_shader_wrapper(self):
 self.update_fill_shader_wrapper()
 return self.fill_shader_wrapper

def update_fill_shader_wrapper(self):
 self.fill_shader_wrapper.vert_data = self.get_fill_shader_data()
 self.fill_shader_wrapper.vert_indices = self.get_triangulation()
 self.fill_shader_wrapper.uniforms = self.get_fill_uniforms()
 self.fill_shader_wrapper.depth_test = self.depth_test

def get_stroke_shader_wrapper(self):
 self.update_stroke_shader_wrapper()
 return self.stroke_shader_wrapper

def update_stroke_shader_wrapper(self):
 self.stroke_shader_wrapper.vert_data = self.get_stroke_shader_data()
 self.stroke_shader_wrapper.uniforms = self.get_stroke_uniforms()
 self.stroke_shader_wrapper.depth_test = self.depth_test

def get_shader_wrapper_list(self):
 # Build up data lists
 fill_shader_wrappers = []
 stroke_shader_wrappers = []
 back_stroke_shader_wrappers = []
 for submob in self.family_members_with_points():
 if submob.has_fill() and not config["use_projection_fill_shaders"]:
 fill_shader_wrappers.append(submob.get_fill_shader_wrapper())

```

```

if submob.has_stroke() and not config["use_projection_stroke_shaders"]:
 ssw = submob.get_stroke_shader_wrapper()
 if submob.draw_stroke_behind_fill:
 back_stroke_shader_wrappers.append(ssw)
 else:
 stroke_shader_wrappers.append(ssw)

Combine data lists
wrapper_lists = [
 back_stroke_shader_wrappers,
 fill_shader_wrappers,
 stroke_shader_wrappers,
]
result = []
for wlist in wrapper_lists:
 if wlist:
 wrapper = wlist[0]
 wrapper.combine_with(*wlist[1:])
 result.append(wrapper)
return result

def get_stroke_uniforms(self):
 result = dict(super().get_shader_uniforms())
 result["joint_type"] = self.joint_type.value
 result["flat_stroke"] = float(self.flat_stroke)
 return result

def get_fill_uniforms(self):
 return {
 "is_fixed_in_frame": float(self.is_fixed_in_frame),
 "is_fixed_orientation": float(self.is_fixed_orientation),
 "fixed_orientation_center": self.fixed_orientation_center,
 "gloss": self.gloss,
 "shadow": self.shadow,
 }

def get_stroke_shader_data(self):
 points = self.points
 if len(self.stroke_data) != len(points):
 self.stroke_data = np.zeros(len(points), dtype=OpenGLVMOBJECT.stroke_dtype)

 if "points" not in self.locked_data_keys:
 nppc = self.n_points_per_curve
 self.stroke_data["point"] = points
 self.stroke_data["prev_point"][:nppc] = points[-nppc:]
 self.stroke_data["prev_point"][nppc:] = points[:-nppc]
 self.stroke_data["next_point"][:-nppc] = points[nppc:]
 self.stroke_data["next_point"][-nppc:] = points[:nppc]

 self.read_data_to_shader(self.stroke_data, "color", "stroke_rgba")
 self.read_data_to_shader(self.stroke_data, "stroke_width", "stroke_width")
 self.read_data_to_shader(self.stroke_data, "unit_normal", "unit_normal")

 return self.stroke_data

def get_fill_shader_data(self):
 points = self.points
 if len(self.fill_data) != len(points):
 self.fill_data = np.zeros(len(points), dtype=OpenGLVMOBJECT.fill_dtype)
 self.fill_data["vert_index"][:, 0] = range(len(points))

 self.read_data_to_shader(self.fill_data, "point", "points")
 self.read_data_to_shader(self.fill_data, "color", "fill_rgba")
 self.read_data_to_shader(self.fill_data, "unit_normal", "unit_normal")

 return self.fill_data

def refresh_shader_data(self):
 self.get_fill_shader_data()
 self.get_stroke_shader_data()

def get_fill_shader_vert_indices(self):
 return self.get_triangulation()

class OpenGLVGroup(OpenGLVMOBJECT):
 """A group of vectorized mobjects.

 This can be used to group multiple :class:`~.OpenGLVMOBJECT` instances together
 in order to scale, move, ... them together.

 Examples

 To add :class:`~.OpenGLVMOBJECT`'s to a :class:`~.OpenGLVGroup`, you can either use the
 :meth:`~.OpenGLVGroup.add` method, or use the `+` and `+=` operators. Similarly, you
 can subtract elements of a OpenGLVGroup via :meth:`~.OpenGLVGroup.remove` method, or
 `-` and `-=` operators:
 .. doctest::
 >>> from manim import config
 >>> original_renderer = config.renderer
 >>> config.renderer = "opengl"

 >>> from manim import Triangle, Square
 >>> from manim.opengl import OpenGLVGroup
 >>> config.renderer
 <RendererType.OPENGL: 'opengl'>
 >>> vg = OpenGLVGroup()
 >>> triangle, square = Triangle(), Square()
 >>> vg.add(triangle)
 OpenGLVGroup(Triangle)
 """

```

```

>>> vg + square # a new OpenGLGroup is constructed
OpenGLGroup(Triangle, Square)
>>> vg # not modified
OpenGLGroup(Triangle)
>>> vg += square # modifies vg
>>> vg
OpenGLGroup(Triangle, Square)
>>> vg.remove(triangle)
OpenGLGroup(Square)
>>> vg - square # a new OpenGLGroup is constructed
OpenGLGroup()
>>> vg # not modified
OpenGLGroup(Square)
>>> vg -= square # modifies vg
>>> vg
OpenGLGroup()

>>> config.renderer = original_renderer

.. manim:: ArcShapeIris
 :save_last_frame:

class ArcShapeIris(Scene):
 def construct(self):
 colors = [DARK_BROWN, BLUE_E, BLUE_D, BLUE_A, TEAL_B, GREEN_B, YELLOW_E]
 radius = [1 + rad * 0.1 for rad in range(len(colors))]

 circles_group = OpenGLGroup()

 # zip(radius, color) makes the iterator [(radius[i], color[i]) for i in range(radius)]
 circles_group.add(*[Circle(radius=rad, stroke_width=10, color=col)
 for rad, col in zip(radius, colors)])
 self.add(circles_group)
 """

def __init__(self, *vmobjects, **kwargs):
 if not all(isinstance(m, OpenGLMobject) for m in vmobjects):
 raise Exception("All submobjects must be of type OpenGLMobject")
 super().__init__(**kwargs)
 self.add(*vmobjects)

def __repr__(self):
 return (
 self.__class__.__name__
 + "("
 + ", ".join(str(mob) for mob in self.submobjects)
 + ")"
)

def __str__(self):
 return (
 f"{self.__class__.__name__} of {len(self.submobjects)} "
 f"subobject{'s' if len(self.submobjects) > 0 else ''}"
)

def add(self, *vmobjects: OpenGLMobject):
 """Checks if all passed elements are an instance of OpenGLMobject and then add them to submobjects

 Parameters

 vmobjects
 List of OpenGLMobject to add

 Returns

 :class:`OpenGLGroup`

 Raises

 TypeError
 If one element of the list is not an instance of OpenGLMobject

 Examples

 .. manim:: AddToOpenGLGroup

 class AddToOpenGLGroup(Scene):
 def construct(self):
 circle_red = Circle(color=RED)
 circle_green = Circle(color=GREEN)
 circle_blue = Circle(color=BLUE)
 circle_red.shift(LEFT)
 circle_blue.shift(RIGHT)
 gr = OpenGLGroup(circle_red, circle_green)
 gr2 = OpenGLGroup(circle_blue) # Constructor uses add directly
 self.add(gr,gr2)
 self.wait()
 gr += gr2 # Add group to another
 self.play(
 gr.animate.shift(DOWN),
)
 gr -= gr2 # Remove group
 self.play(# Animate groups separately
 gr.animate.shift(LEFT),
 gr2.animate.shift(UP),
)
 self.play(#Animate groups without modification
 (gr+gr2).animate.shift(RIGHT)
)
 self.play(# Animate group without component
 (gr-circle_red).animate.shift(RIGHT)
)
 """

```

```

if not all(isinstance(m, OpenGLVMOBJ) for m in vmobjects):
 raise TypeError("All subobjects must be of type OpenGLVMOBJ")
return super().add(*vmobjects)

def __add__(self, vmobject):
 return OpenGLVGroup(*self.subobjects, vmobject)

def __iadd__(self, vmobject):
 return self.add(vmobject)

def __sub__(self, vmobject):
 copy = OpenGLVGroup(*self.subobjects)
 copy.remove(vmobject)
 return copy

def __isub__(self, vmobject):
 return self.remove(vmobject)

def __setitem__(self, key: int, value: OpenGLVMOBJ | Sequence[OpenGLVMOBJ]):
 """Override the [] operator for item assignment.

Parameters

key
 The index of the subobject to be assigned
value
 The vmobject value to assign to the key

Returns

None

Tests

.. doctest::

 >>> from manim import config
 >>> original_renderer = config.renderer
 >>> config.renderer = "opengl"

 >>> vgroup = OpenGLVGroup(OpenGLVMOBJ())
 >>> new_obj = OpenGLVMOBJ()
 >>> vgroup[0] = new_obj

 >>> config.renderer = original_renderer
"""
if not all(isinstance(m, OpenGLVMOBJ) for m in value):
 raise TypeError("All subobjects must be of type OpenGLVMOBJ")
self.subobjects[key] = value

class OpenGLVectorizedPoint(OpenGLPoint, OpenGLVMOBJ):
 def __init__(
 self,
 location=ORIGIN,
 color=BLACK,
 fill_opacity=0,
 stroke_width=0,
 artificial_width=0.01,
 artificial_height=0.01,
 **kwargs,
):
 self.artificial_width = artificial_width
 self.artificial_height = artificial_height

 super().__init__(
 color=color, fill_opacity=fill_opacity, stroke_width=stroke_width, **kwargs
)
 self.set_points(np.array([location]))

class OpenGLCurvesAsSubobjects(OpenGLVGroup):
 """Convert a curve's elements to subobjects.

Examples

.. manim:: LineGradientExample
 :save_last_frame:

 class LineGradientExample(Scene):
 def construct(self):
 curve = ParametricFunction(lambda t: [t, np.sin(t), 0], t_range=[-PI, PI, 0.01], stroke_width=10)
 new_curve = CurvesAsSubobjects(curve)
 new_curve.set_color_by_gradient(BLUE, RED)
 self.add(new_curve.shift(UP), curve)

"""

def __init__(self, vmobject, **kwargs):
 super().__init__(**kwargs)
 for tup in vmobject.get_bezier_tuples():
 part = OpenGLVMOBJ()
 part.set_points(tup)
 part.match_style(vmobject)
 self.add(part)

class OpenGLDashedVMObject(OpenGLVMOBJ):
 """A :class:`OpenGLVMOBJ` composed of dashes instead of lines.

Examples

```

```

.. manim:: DashedVMobjectExample
 :save_last_frame:

class DashedVMobjectExample(Scene):
 def construct(self):
 r = 0.5

 top_row = OpenGLVGGroup() # Increasing num_dashes
 for dashes in range(2, 12):
 circ = DashedVMobject(Circle(radius=r, color=WHITE), num_dashes=dashes)
 top_row.add(circ)

 middle_row = OpenGLVGGroup() # Increasing dashed_ratio
 for ratio in np.arange(1 / 11, 1, 1 / 11):
 circ = DashedVMobject(
 Circle(radius=r, color=WHITE), dashed_ratio=ratio
)
 middle_row.add(circ)

 sq = DashedVMobject(Square(1.5, color=RED))
 penta = DashedVMobject(RegularPolygon(5, color=BLUE))
 bottom_row = OpenGLVGGroup(sq, penta)

 top_row.arrange(buff=0.4)
 middle_row.arrange()
 bottom_row.arrange(buff=1)
 everything = OpenGLVGGroup(top_row, middle_row, bottom_row).arrange(DOWN, buff=1)
 self.add(everything)
 """

def __init__(
 self,
 vmobject: OpenGLMobject,
 num_dashes: int = 15,
 dashed_ratio: float = 0.5,
 color: Color = WHITE,
 **kwargs,
):
 self.dashed_ratio = dashed_ratio
 self.num_dashes = num_dashes
 super().__init__(color=color, **kwargs)
 r = self.dashed_ratio
 n = self.num_dashes
 if num_dashes > 0:
 # Assuming total length is 1
 dash_len = r / n
 if vmobject.is_closed():
 void_len = (1 - r) / n
 else:
 void_len = (1 - r) / (n - 1)

 self.add(
 *(
 vmobject.get_subcurve(
 i * (dash_len + void_len),
 i * (dash_len + void_len) + dash_len,
)
 for i in range(n)
)
)
 # Family is already taken care of by get_subcurve
 # implementation
 self.match_style(vmobject, recurse=False)

```

# mobject\_update\_utils.py

```
"""Utility functions for continuous animation of mobjects."""
from __future__ import annotations

__all__ = [
 "assert_is_mobject_method",
 "always",
 "f_always",
 "always_redraw",
 "always_shift",
 "always_rotate",
 "turn_animation_into_updater",
 "cycle_animation",
]
import inspect
import numpy as np
from manim.constants import DEGREES, RIGHT
from manim.mobject.mobject import Mobject
from manim.opengl import OpenGLMobject

def assert_is_mobject_method(method):
 assert inspect.ismethod(method)
 mobobject = method.__self__
 assert isinstance(mobobject, (Mobject, OpenGLMobject))

def always(method, *args, **kwargs):
 assert_is_mobject_method(method)
 mobobject = method.__self__
 func = method.__func__
 mobobject.add_updater(lambda m: func(m, *args, **kwargs))
 return mobobject

def f_always(method, *arg_generators, **kwargs):
 """
 More functional version of always, where instead
 of taking in args, it takes in functions which output
 the relevant arguments.
 """
 assert_is_mobject_method(method)
 mobobject = method.__self__
 func = method.__func__

 def updater(mob):
 args = [arg_generator() for arg_generator in arg_generators]
 func(mob, *args, **kwargs)

 mobobject.add_updater(updater)
 return mobobject

def always_redraw(func):
 mob = func()
 mob.add_updater(lambda m: mob.become(func()))
 return mob

def always_shift(mobject, direction=RIGHT, rate=0.1):
 def normalize(v):
 norm = np.linalg.norm(v)
 if norm == 0:
 return v
 return v / norm

 mobject.add_updater(lambda m, dt: m.shift(dt * rate * normalize(direction)))
 return mobject

def always_rotate(mobject, rate=20 * DEGREES, **kwargs):
 mobject.add_updater(lambda m, dt: m.rotate(dt * rate, **kwargs))
 return mobject

def turn_animation_into_updater(animation, cycle=False, **kwargs):
 """
```

```
Add an updater to the animation's mobject which applies
the interpolation and update functions of the animation
```

```
If cycle is True, this repeats over and over. Otherwise,
the updater will be popped upon completion
```

```
"""
mobject = animation.mobject
animation.suspend_mobject_updating = False
animation.begin()
animation.total_time = 0

def update(m, dt):
 run_time = animation.get_run_time()
 time_ratio = animation.total_time / run_time
 if cycle:
 alpha = time_ratio % 1
 else:
 alpha = np.clip(time_ratio, 0, 1)
 if alpha >= 1:
 animation.finish()
 m.remove_updater(update)
 return
 animation.interpolate(alpha)
 animation.update_mobjects(dt)
 animation.total_time += dt

mobject.add_updater(update)
return mobject
```

```
def cycle_animation(animation, **kwargs):
 return turn_animation_into_updater(animation, cycle=True, **kwargs)
```

# section.py

```
"""building blocks of segmented video API"""

from __future__ import annotations

from enum import Enum
from pathlib import Path
from typing import Any

from manim import get_video_metadata

class DefaultSectionType(str, Enum):
 """The type of a section can be used for third party applications.
 A presentation system could for example use the types to created loops.

 Examples

 This class can be reimplemented for more types::

 class PresentationSectionType(str, Enum):
 # start, end, wait for continuation by user
 NORMAL = "presentation.normal"
 # start, end, immediately continue to next section
 SKIP = "presentation.skip"
 # start, end, restart, immediately continue to next section when continued by user
 LOOP = "presentation.loop"
 # start, end, restart, finish animation first when user continues
 COMPLETE_LOOP = "presentation.complete_loop"
 """

 NORMAL = "default.normal"

class Section:
 """A :class:`Scene` can be segmented into multiple Sections.
 Refer to :doc:`the documentation</tutorials/output_and_config>` for more info.
 It consists of multiple animations.

 Attributes

 type
 Can be used by a third party applications to classify different types of sections.
 video
 Path to video file with animations belonging to section relative to sections directory.
 If ``None``, then the section will not be saved.
 name
 Human readable, non-unique name for this section.
 skip_animations
 Skip rendering the animations in this section when ``True``.
 partial_movie_files
 Animations belonging to this section.

 See Also

 :class:`DefaultSectionType`
 :meth:`CairoRenderer.update_skipping_status`
 :meth:`OpenGLRenderer.update_skipping_status`
 """

 def __init__(self, type: str, video: str | None, name: str, skip_animations: bool):
 self.type = type
 # None when not to be saved -> still keeps section alive
 self.video: str | None = video
 self.name = name
 self.skip_animations = skip_animations
 self.partial_movie_files: list[str | None] = []

 def is_empty(self) -> bool:
 """Check whether this section is empty.

 Note that animations represented by ``None`` are also counted.
 """
 return len(self.partial_movie_files) == 0

 def get_clean_partial_movie_files(self) -> list[str]:
 """Return all partial movie files that are not ``None``."""
 return [el for el in self.partial_movie_files if el is not None]

 def get_dict(self, sections_dir: Path) -> dict[str, Any]:
 """Get dictionary representation with metadata of output video.
```

```
The output from this function is used from every section to build the sections index file.
The output video must have been created in the ``sections_dir`` before executing this method.
This is the main part of the Segmented Video API.
"""
if self.video is None:
 raise ValueError(
 f"Section '{self.name}' cannot be exported as dict, it does not have a video path assigned to it"
)

video_metadata = get_video_metadata(sections_dir / self.video)
return dict(
 {
 "name": self.name,
 "type": self.type,
 "video": self.video,
 },
 **video_metadata,
)

def __repr__(self):
 return f"<Section '{self.name}' stored in '{self.video}'>"
```

# moving\_camera.py

```
"""A camera able to move through a scene.

.. SEEALSO::

 :mod:`.moving_camera_scene`

"""

from __future__ import annotations

__all__ = ["MovingCamera"]

import numpy as np

from .. import config
from ..camera.camera import Camera
from ..constants import DOWN, LEFT, RIGHT, UP
from ..mobject.frame import ScreenRectangle
from ..mobject.mobject import Mobject
from ..utils.color import WHITE

class MovingCamera(Camera):
 """
 Stays in line with the height, width and position of it's 'frame', which is a Rectangle

 .. SEEALSO::

 :class:`.MovingCameraScene`

 """

 def __init__(self,
 frame=None,
 fixed_dimension=0, # width
 default_frame_stroke_color=WHITE,
 default_frame_stroke_width=0,
 **kwargs,
):
 """
 Frame is a Mobject, (should almost certainly be a rectangle)
 determining which region of space the camera displays
 """
 self.fixed_dimension = fixed_dimension
 self.default_frame_stroke_color = default_frame_stroke_color
 self.default_frame_stroke_width = default_frame_stroke_width
 if frame is None:
 frame = ScreenRectangle(height=config["frame_height"])
 frame.set_stroke(
 self.default_frame_stroke_color,
 self.default_frame_stroke_width,
)
 self.frame = frame
 super().__init__(**kwargs)

 # TODO, make these work for a rotated frame
 @property
 def frame_height(self):
 """Returns the height of the frame.

 Returns

 float
 The height of the frame.
 """
 return self.frame.height

 @property
 def frame_width(self):
 """Returns the width of the frame

 Returns

 float
 The width of the frame.
 """
 return self.frame.width

 @property
 def frame_center(self):
```

```

 """Returns the centerpoint of the frame in cartesian coordinates.

 Returns

 np.array
 The cartesian coordinates of the center of the frame.
 """
 return self.frame.get_center()

@frame_height.setter
def frame_height(self, frame_height: float):
 """Sets the height of the frame in MUnits.

 Parameters

 frame_height
 The new frame_height.
 """
 self.frame.stretch_to_fit_height(frame_height)

@frame_width.setter
def frame_width(self, frame_width: float):
 """Sets the width of the frame in MUnits.

 Parameters

 frame_width
 The new frame_width.
 """
 self.frame.stretch_to_fit_width(frame_width)

@frame_center.setter
def frame_center(self, frame_center: np.ndarray | list | tuple | Mobject):
 """Sets the centerpoint of the frame.

 Parameters

 frame_center
 The point to which the frame must be moved.
 If is of type mobject, the frame will be moved to
 the center of that mobject.
 """
 self.frame.move_to(frame_center)

def capture_mobjects(self, mobjects, **kwargs):
 # self.reset_frame_center()
 # self.realign_frame_shape()
 super().capture_mobjects(mobjects, **kwargs)

Since the frame can be moving around, the cairo
context used for updating should be regenerated
at each frame. So no caching.
def get_cached_cairo_context(self, pixel_array):
 """
 Since the frame can be moving around, the cairo
 context used for updating should be regenerated
 at each frame. So no caching.
 """
 return None

def cache_cairo_context(self, pixel_array, ctx):
 """
 Since the frame can be moving around, the cairo
 context used for updating should be regenerated
 at each frame. So no caching.
 """
 pass

def reset_frame_center(self):
self.frame_center = self.frame.get_center()

def realign_frame_shape(self):
height, width = self.frame_shape
if self.fixed_dimension == 0:
self.frame_shape = (height, self.frame.width)
else:
self.frame_shape = (self.frame.height, width)
self.resize_frame_shape(fixed_dimension=self.fixed_dimension)

def get_mobjects_indicating_movement(self):
 """
 Returns all mobjects whose movement implies that the camera
 should think of all other mobjects on the screen as moving

 Returns

 list
 """

```

```

"""
return [self.frame]

def auto_zoom(
 self,
 mobjects: list[Mobject],
 margin: float = 0,
 only_mobjects_in_frame: bool = False,
 animate: bool = True,
):
 """Zooms on to a given array of mobjects (or a singular mobject)
 and automatically resizes to frame all the mobjects.

 .. NOTE::

 This method only works when 2D-objects in the XY-plane are considered, it
 will not work correctly when the camera has been rotated.

 Parameters

 mobjects
 The mobject or array of mobjects that the camera will focus on.

 margin
 The width of the margin that is added to the frame (optional, 0 by default).

 only_mobjects_in_frame
 If set to ``True``, only allows focusing on mobjects that are already in frame.

 animate
 If set to ``False``, applies the changes instead of returning the corresponding animation

 Returns

 Union[_AnimationBuilder, ScreenRectangle]
 _AnimationBuilder that zooms the camera view to a given list of mobjects
 or ScreenRectangle with position and size updated to zoomed position.

"""

scene_critical_x_left = None
scene_critical_x_right = None
scene_critical_y_up = None
scene_critical_y_down = None

for m in mobjects:
 if (m == self.frame) or (
 only_mobjects_in_frame and not self.is_in_frame(m)
):
 # detected camera frame, should not be used to calculate final position of camera
 continue

 # initialize scene critical points with first mobjects critical points
 if scene_critical_x_left is None:
 scene_critical_x_left = m.get_critical_point(LEFT)[0]
 scene_critical_x_right = m.get_critical_point(RIGHT)[0]
 scene_critical_y_up = m.get_critical_point(UP)[1]
 scene_critical_y_down = m.get_critical_point(DOWN)[1]

 else:
 if m.get_critical_point(LEFT)[0] < scene_critical_x_left:
 scene_critical_x_left = m.get_critical_point(LEFT)[0]

 if m.get_critical_point(RIGHT)[0] > scene_critical_x_right:
 scene_critical_x_right = m.get_critical_point(RIGHT)[0]

 if m.get_critical_point(UP)[1] > scene_critical_y_up:
 scene_critical_y_up = m.get_critical_point(UP)[1]

 if m.get_critical_point(DOWN)[1] < scene_critical_y_down:
 scene_critical_y_down = m.get_critical_point(DOWN)[1]

 # calculate center x and y
 x = (scene_critical_x_left + scene_critical_x_right) / 2
 y = (scene_critical_y_up + scene_critical_y_down) / 2

 # calculate proposed width and height of zoomed scene
 new_width = abs(scene_critical_x_left - scene_critical_x_right)
 new_height = abs(scene_critical_y_up - scene_critical_y_down)

 m_target = self.frame.animate if animate else self.frame
 # zoom to fit all mobjects along the side that has the largest size
 if new_width / self.frame.width > new_height / self.frame.height:
 return m_target.set_x(x).set_y(y).set(width=new_width + margin)
 else:
 return m_target.set_x(x).set_y(y).set(height=new_height + margin)

```

# opengl.py

```
from __future__ import annotations

import numpy as np
import numpy.linalg as linalg

from .. import config

depth = 20

def matrix_to_shader_input(matrix):
 return tuple(matrix.T.ravel())

def orthographic_projection_matrix(
 width=None,
 height=None,
 near=1,
 far=depth + 1,
 format=True,
):
 if width is None:
 width = config["frame_width"]
 if height is None:
 height = config["frame_height"]
 projection_matrix = np.array(
 [
 [2 / width, 0, 0, 0],
 [0, 2 / height, 0, 0],
 [0, 0, -2 / (far - near), -(far + near) / (far - near)],
 [0, 0, 0, 1],
],
)
 if format:
 return matrix_to_shader_input(projection_matrix)
 else:
 return projection_matrix

def perspective_projection_matrix(width=None, height=None, near=2, far=50, format=True):
 if width is None:
 width = config["frame_width"] / 6
 if height is None:
 height = config["frame_height"] / 6
 projection_matrix = np.array(
 [
 [2 * near / width, 0, 0, 0],
 [0, 2 * near / height, 0, 0],
 [0, 0, (far + near) / (near - far), (2 * far * near) / (near - far)],
 [0, 0, -1, 0],
],
)
 if format:
 return matrix_to_shader_input(projection_matrix)
 else:
 return projection_matrix

def translation_matrix(x=0, y=0, z=0):
 return np.array(
 [
 [1, 0, 0, x],
 [0, 1, 0, y],
 [0, 0, 1, z],
 [0, 0, 0, 1],
],
)

def x_rotation_matrix(x=0):
 return np.array(
 [
 [1, 0, 0, 0],
 [0, np.cos(x), -np.sin(x), 0],
 [0, np.sin(x), np.cos(x), 0],
 [0, 0, 0, 1],
],
)

def y_rotation_matrix(y=0):
```

```

return np.array(
 [
 [np.cos(y), 0, np.sin(y), 0],
 [0, 1, 0, 0],
 [-np.sin(y), 0, np.cos(y), 0],
 [0, 0, 0, 1],
],
)

def z_rotation_matrix(z=0):
 return np.array(
 [
 [np.cos(z), -np.sin(z), 0, 0],
 [np.sin(z), np.cos(z), 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1],
],
)

TODO: When rotating around the x axis, rotation eventually stops.
def rotate_in_place_matrix(initial_position, x=0, y=0, z=0):
 return np.matmul(
 translation_matrix(*-initial_position),
 np.matmul(
 rotation_matrix(x, y, z),
 translation_matrix(*initial_position),
),
)

def rotation_matrix(x=0, y=0, z=0):
 return np.matmul(
 np.matmul(x_rotation_matrix(x), y_rotation_matrix(y)),
 z_rotation_matrix(z),
)

def scale_matrix(scale_factor=1):
 return np.array(
 [
 [scale_factor, 0, 0, 0],
 [0, scale_factor, 0, 0],
 [0, 0, scale_factor, 0],
 [0, 0, 0, 1],
],
)

def view_matrix(
 translation=None,
 x_rotation=0,
 y_rotation=0,
 z_rotation=0,
):
 if translation is None:
 translation = np.array([0, 0, depth / 2 + 1])
 model_matrix = np.matmul(
 np.matmul(
 translation_matrix(*translation),
 rotation_matrix(x=x_rotation, y=y_rotation, z=z_rotation),
),
 scale_matrix(),
)
 return tuple(linalg.inv(model_matrix).T.ravel())

```

# color.py

```
"""Colors and utility functions for conversion between different color models."""

from __future__ import annotations

__all__ = [
 "color_to_rgb",
 "color_to_rgba",
 "rgb_to_color",
 "rgba_to_color",
 "rgb_to_hex",
 "hex_to_rgb",
 "invert_color",
 "color_to_int_rgb",
 "color_to_int_rgba",
 "color_gradient",
 "interpolate_color",
 "average_color",
 "random_bright_color",
 "random_color",
 "get_shaded_rgb",
]
import random
from enum import Enum
from typing import Iterable

import numpy as np
from colour import Color

from ..utils.bezier import interpolate
from ..utils.space_ops import normalize

class Colors(Enum):
 """A list of pre-defined colors.

 Examples

 .. manim:: ColorsOverview
 :save_last_frame:
 :hide_source:

 from manim.utils.color import Colors
 class ColorsOverview(Scene):
 def construct(self):
 def color_group(color):
 group = VGroup(
 *[Line(ORIGIN, RIGHT * 1.5, stroke_width=35, color=Colors[name].value)
 for name in subnames(color)]
).arrange_submobjects(buff=0.4, direction=DOWN)

 name = Text(color).scale(0.6).next_to(group, UP, buff=0.3)
 if any(decender in color for decender in "gjpqy"):
 name.shift(DOWN * 0.08)
 group.add(name)
 return group

 def subnames(name):
 return [name + "_" + char for char in "abcde"]

 color_groups = VGroup(
 *[color_group(color)
 for color in [
 "blue",
 "teal",
 "green",
 "yellow",
 "gold",
 "red",
 "maroon",
 "purple",
]
]
).arrange_submobjects(buff=0.2, aligned_edge=DOWN)

 for line, char in zip(color_groups[0], "abcde"):
 color_groups.add(Text(char).scale(0.6).next_to(line, LEFT, buff=0.2))

 """
]
```

```

def named_lines_group(length, colors, names, text_colors, align_to_block):
 lines = VGroup(
 *[
 Line(
 ORIGIN,
 RIGHT * length,
 stroke_width=55,
 color=Colors[color].value,
)
 for color in colors
]
).arrange_submobjects(buff=0.6, direction=DOWN)

 for line, name, color in zip(lines, names, text_colors):
 line.add(Text(name, color=color).scale(0.6).move_to(line))
 lines.next_to(color_groups, DOWN, buff=0.5).align_to(
 color_groups[align_to_block], LEFT
)
 return lines

other_colors = (
 "pink",
 "light_pink",
 "orange",
 "light_brown",
 "dark_brown",
 "gray_brown",
)
other_lines = named_lines_group(
 3.2,
 other_colors,
 other_colors,
 [BLACK] * 4 + [WHITE] * 2,
 0,
)
gray_lines = named_lines_group(
 6.6,
 ["white"] + subnames("gray") + ["black"],
 [
 "white",
 "lighter_gray / gray_a",
 "light_gray / gray_b",
 "gray / gray_c",
 "dark_gray / gray_d",
 "darker_gray / gray_e",
 "black",
],
 [BLACK] * 3 + [WHITE] * 4,
 2,
)
pure_colors = (
 "pure_red",
 "pure_green",
 "pure_blue",
)
pure_lines = named_lines_group(
 3.2,
 pure_colors,
 pure_colors,
 [BLACK, BLACK, WHITE],
 6,
)
self.add(color_groups, other_lines, gray_lines, pure_lines)
VGroup(*self.mobjects).move_to(ORIGIN)

```

The preferred way of using these colors is by importing their constants from manim:

```
.. code-block:: pycon
```

```
>>> from manim import RED, GREEN, BLUE
>>> RED
'#FC6255'
```

Note this way uses the name of the colors in UPPERCASE.

Alternatively, you can also import this Enum directly and use its members directly, through the use of :code:`color.value`. Note this way uses the name of the colors in lowercase.

```

.. code-block:: pycon

 >>> from manim.utils.color import Colors
 >>> Colors.red.value
 '#FC6255'

.. note::

 The colors of type "C" have an alias equal to the colorname without a letter,
 e.g. GREEN = GREEN_C

"""

white: str = "#FFFFFF"
gray_a: str = "#DDDDDD"
gray_b: str = "#BBBBBB"
gray_c: str = "#888888"
gray_d: str = "#444444"
gray_e: str = "#222222"
black: str = "#000000"
lighter_gray: str = gray_a
light_gray: str = gray_b
gray: str = gray_c
dark_gray: str = gray_d
darker_gray: str = gray_e

blue_a: str = "#C7E9F1"
blue_b: str = "#9CDCEB"
blue_c: str = "#58C4DD"
blue_d: str = "#29ABCA"
blue_e: str = "#236B8E"
pure_blue: str = "#0000FF"
blue: str = blue_c
dark_blue: str = blue_e

teal_a: str = "#ACEAD7"
teal_b: str = "#76DDC0"
teal_c: str = "#5CD0B3"
teal_d: str = "#55C1A7"
teal_e: str = "#49A88F"
teal: str = teal_c

green_a: str = "#C9E2AE"
green_b: str = "#A6CF8C"
green_c: str = "#83C167"
green_d: str = "#77B05D"
green_e: str = "#699C52"
pure_green: str = "#00FF00"
green: str = green_c

yellow_a: str = "#FFF1B6"
yellow_b: str = "#FFEA94"
yellow_c: str = "#FFFF00"
yellow_d: str = "#F4D345"
yellow_e: str = "#E8C11C"
yellow: str = yellow_c

gold_a: str = "#F7C797"
gold_b: str = "#F9B775"
gold_c: str = "#F0AC5F"
gold_d: str = "#E1A158"
gold_e: str = "#C78D46"
gold: str = gold_c

red_a: str = "#F7A1A3"
red_b: str = "#FF8080"
red_c: str = "#FC6255"
red_d: str = "#E65A4C"
red_e: str = "#CF5044"
pure_red: str = "#FF0000"
red: str = red_c

maroon_a: str = "#ECABC1"
maroon_b: str = "#EC92AB"
maroon_c: str = "#C55F73"
maroon_d: str = "#A24D61"
maroon_e: str = "#94424F"
maroon: str = maroon_c

purple_a: str = "#CAA3E8"
purple_b: str = "#B189C6"
purple_c: str = "#9A72AC"
purple_d: str = "#715582"
purple_e: str = "#644172"
purple: str = purple_c

pink: str = "#D147BD"

```

```

light_pink: str = "#DC75CD"

orange: str = "#FF862F"
light_brown: str = "#CD853F"
dark_brown: str = "#8B4513"
gray_brown: str = "#736357"

def print_constant_definitions():
"""
A simple function used to generate the constant values below. To run it
paste this function and the Colors class into a file and run them.
"""

constants_names: list[str] = []
for name in Colors.__members__.keys():
 name_upper = name.upper()

 constants_names.append(name_upper)
 print(f"{name_upper} = Colors.{name}")

 if "GRAY" in name_upper:
 name_upper = name_upper.replace("GRAY", "GREY")

 constants_names.append(name_upper)
 print(f"{name_upper} = Colors.{name}")

constants_names_repr = '[\n "' + '\n "'.join(constants_names) + '",\n]'

print(f"\n__all__ += {constants_names_repr}")

WHITE: str = "#FFFFFF"
GRAY_A: str = "#DDDDDD"
GREY_A: str = "#DDDDDD"
GRAY_B: str = "#BBBBBB"
GREY_B: str = "#BBBBBB"
GRAY_C: str = "#888888"
GREY_C: str = "#888888"
GRAY_D: str = "#444444"
GREY_D: str = "#444444"
GRAY_E: str = "#222222"
GREY_E: str = "#222222"
BLACK: str = "#000000"
LIGHTER_GRAY: str = "#DDDDDD"
LIGHTER_GREY: str = "#DDDDDD"
LIGHT_GRAY: str = "#BBBBBB"
LIGHT_GREY: str = "#BBBBBB"
GRAY: str = "#888888"
GREY: str = "#888888"
DARK_GRAY: str = "#444444"
DARK_GREY: str = "#444444"
DARKER_GRAY: str = "#222222"
DARKER_GREY: str = "#222222"
BLUE_A: str = "#C7E9F1"
BLUE_B: str = "#9CDCEB"
BLUE_C: str = "#58C4DD"
BLUE_D: str = "#29ABCA"
BLUE_E: str = "#236B8E"
PURE_BLUE: str = "#0000FF"
BLUE: str = "#58C4DD"
DARK_BLUE: str = "#236B8E"
TEAL_A: str = "#ACEAD7"
TEAL_B: str = "#76DC00"
TEAL_C: str = "#5CD0B3"
TEAL_D: str = "#55C1A7"
TEAL_E: str = "#49A88F"
TEAL: str = "#5CD0B3"
GREEN_A: str = "#C9E2AE"
GREEN_B: str = "#A6CF8C"
GREEN_C: str = "#83C167"
GREEN_D: str = "#77B05D"
GREEN_E: str = "#699C52"
PURE_GREEN: str = "#00FF00"
GREEN: str = "#83C167"
YELLOW_A: str = "#FFF1B6"
YELLOW_B: str = "#FFEA94"
YELLOW_C: str = "#FFFF00"
YELLOW_D: str = "#F4D345"
YELLOW_E: str = "#E8C11C"
YELLOW: str = "#FFFF00"
GOLD_A: str = "#F7C797"
GOLD_B: str = "#F9B775"
GOLD_C: str = "#FOAC5F"
GOLD_D: str = "#E1A158"
GOLD_E: str = "#C78D46"
GOLD: str = "#FOAC5F"
RED_A: str = "#F7A1A3"

```

```
RED_B: str = "#FF8080"
RED_C: str = "#FC6255"
RED_D: str = "#E65A4C"
RED_E: str = "#CF5044"
PURE_RED: str = "#FF0000"
RED: str = "#FC6255"
MAROON_A: str = "#ECABC1"
MAROON_B: str = "#EC92AB"
MAROON_C: str = "#C55F73"
MAROON_D: str = "#A24D61"
MAROON_E: str = "#94424F"
MAROON: str = "#C55F73"
PURPLE_A: str = "#CAA3E8"
PURPLE_B: str = "#B189C6"
PURPLE_C: str = "#9A72AC"
PURPLE_D: str = "#715582"
PURPLE_E: str = "#644172"
PURPLE: str = "#9A72AC"
PINK: str = "#D147BD"
LIGHT_PINK: str = "#DC75CD"
ORANGE: str = "#FF862F"
LIGHT_BROWN: str = "#CD853F"
DARK_BROWN: str = "#8B4513"
GRAY_BROWN: str = "#736357"
GREY_BROWN: str = "#736357"

all += [
 "WHITE",
 "GRAY_A",
 "GRAY_A",
 "GRAY_B",
 "GRAY_B",
 "GRAY_C",
 "GRAY_C",
 "GRAY_D",
 "GRAY_D",
 "GRAY_E",
 "GRAY_E",
 "BLACK",
 "LIGHTER_GRAY",
 "LIGHTER_GREY",
 "LIGHT_GRAY",
 "LIGHT_GREY",
 "GRAY",
 "GRAY",
 "DARK_GRAY",
 "DARK_GREY",
 "DARKER_GRAY",
 "DARKER_GREY",
 "BLUE_A",
 "BLUE_B",
 "BLUE_C",
 "BLUE_D",
 "BLUE_E",
 "PURE_BLUE",
 "BLUE",
 "DARK_BLUE",
 "TEAL_A",
 "TEAL_B",
 "TEAL_C",
 "TEAL_D",
 "TEAL_E",
 "TEAL",
 "GREEN_A",
 "GREEN_B",
 "GREEN_C",
 "GREEN_D",
 "GREEN_E",
 "PURE_GREEN",
 "GREEN",
 "YELLOW_A",
 "YELLOW_B",
 "YELLOW_C",
 "YELLOW_D",
 "YELLOW_E",
 "YELLOW",
 "GOLD_A",
 "GOLD_B",
 "GOLD_C",
 "GOLD_D",
 "GOLD_E",
 "GOLD",
 "RED_A",
 "RED_B",
 "RED_C",
 "RED_D",
 "RED_E",
```

```

 "PURE_RED",
 "RED",
 "MAROON_A",
 "MAROON_B",
 "MAROON_C",
 "MAROON_D",
 "MAROON_E",
 "MAROON",
 "PURPLE_A",
 "PURPLE_B",
 "PURPLE_C",
 "PURPLE_D",
 "PURPLE_E",
 "PURPLE",
 "PINK",
 "LIGHT_PINK",
 "ORANGE",
 "LIGHT_BROWN",
 "DARK_BROWN",
 "GRAY_BROWN",
 "GREY_BROWN",
]
]

def color_to_rgb(color: Color | str) -> np.ndarray:
 if isinstance(color, str):
 return hex_to_rgb(color)
 elif isinstance(color, Color):
 return np.array(color.get_rgb())
 else:
 raise ValueError("Invalid color type: " + str(color))

def color_to_rgba(color: Color | str, alpha: float = 1) -> np.ndarray:
 return np.array([*color_to_rgb(color), alpha])

def rgb_to_color(rgb: Iterable[float]) -> Color:
 return Color(rgb=rgb)

def rgba_to_color(rgba: Iterable[float]) -> Color:
 return rgb_to_color(rgba[:3])

def rgb_to_hex(rgb: Iterable[float]) -> str:
 return "#" + "".join("%02x" % round(255 * x) for x in rgb)

def hex_to_rgb(hex_code: str) -> np.ndarray:
 hex_part = hex_code[1:]
 if len(hex_part) == 3:
 hex_part = "".join([2 * c for c in hex_part])
 return np.array([int(hex_part[i : i + 2], 16) / 255 for i in range(0, 6, 2)])

def invert_color(color: Color) -> Color:
 return rgb_to_color(1.0 - color_to_rgb(color))

def color_to_int_rgb(color: Color) -> np.ndarray:
 return (255 * color_to_rgb(color)).astype("uint8")

def color_to_int_rgba(color: Color, opacity: float = 1.0) -> np.ndarray:
 alpha_multiplier = np.vectorize(lambda x: int(x * opacity))
 return alpha_multiplier(np.append(color_to_int_rgb(color), 255))

def color_gradient(
 reference_colors: Iterable[Color],
 length_of_output: int,
) -> list[Color]:
 if length_of_output == 0:
 return reference_colors[0]
 rgbs = list(map(color_to_rgb, reference_colors))
 alphas = np.linspace(0, (len(rgbs) - 1), length_of_output)
 floors = alphas.astype("int")
 alphas_mod1 = alphas % 1
 # End edge case
 alphas_mod1[-1] = 1
 floors[-1] = len(rgbs) - 2
 return [
 rgb_to_color(interpolate(rgbs[i], rgbs[i + 1], alpha))
 for i, alpha in zip(floors, alphas_mod1)
]

```

```
def interpolate_color(color1: Color, color2: Color, alpha: float) -> Color:
 rgb = interpolate(color_to_rgb(color1), color_to_rgb(color2), alpha)
 return rgb_to_color(rgb)

def average_color(*colors: Color) -> Color:
 rgbs = np.array(list(map(color_to_rgb, colors)))
 mean_rgb = np.apply_along_axis(np.mean, 0, rgbs)
 return rgb_to_color(mean_rgb)

def random_bright_color() -> Color:
 color = random_color()
 curr_rgb = color_to_rgb(color)
 new_rgb = interpolate(curr_rgb, np.ones(len(curr_rgb)), 0.5)
 return Color(rgb=new_rgb)

def random_color() -> Color:
 return random.choice([c.value for c in list(Colors)])

def get_shaded_rgb(
 rgb: np.ndarray,
 point: np.ndarray,
 unit_normal_vect: np.ndarray,
 light_source: np.ndarray,
) -> np.ndarray:
 to_sun = normalize(light_source - point)
 factor = 0.5 * np.dot(unit_normal_vect, to_sun) ** 3
 if factor < 0:
 factor *= 0.5
 result = rgb + factor
 return result
```

# images.py

```
"""Image manipulation utilities."""

from __future__ import annotations

__all__ = [
 "get_full_raster_image_path",
 "drag_pixels",
 "invert_image",
 "change_to_rgba_array",
]
from pathlib import Path

import numpy as np
from PIL import Image

from .. import config
from ..utils.file_ops import seek_full_path_from_defaults

def get_full_raster_image_path(image_file_name: str) -> Path:
 return seek_full_path_from_defaults(
 image_file_name,
 default_dir=config.get_dir("assets_dir"),
 extensions=[".jpg", ".jpeg", ".png", ".gif", ".ico"],
)

def get_full_vector_image_path(image_file_name: str) -> Path:
 return seek_full_path_from_defaults(
 image_file_name,
 default_dir=config.get_dir("assets_dir"),
 extensions=[".svg"],
)

def drag_pixels(frames: list[np.array]) -> list[np.array]:
 curr = frames[0]
 new_frames = []
 for frame in frames:
 curr += (curr == 0) * np.array(frame)
 new_frames.append(np.array(curr))
 return new_frames

def invert_image(image: np.array) -> Image:
 arr = np.array(image)
 arr = (255 * np.ones(arr.shape)).astype(arr.dtype) - arr
 return Image.fromarray(arr)

def change_to_rgba_array(image, dtype="uint8"):
 """Converts an RGB array into RGBA with the alpha value opacity maxed."""
 pa = image
 if len(pa.shape) == 2:
 pa = pa.reshape(list(pa.shape) + [1])
 if pa.shape[2] == 1:
 pa = pa.repeat(3, axis=2)
 if pa.shape[2] == 3:
 alphas = 255 * np.ones(
 list(pa.shape[:2]) + [1],
 dtype=dtype,
)
 pa = np.append(pa, alphas, axis=2)
 return pa
```

# ipython\_magic.py

```
"""Utilities for using Manim with IPython (in particular: Jupyter notebooks)"""
from __future__ import annotations

import mimetypes
import os
import shutil
from datetime import datetime
from pathlib import Path
from typing import Any

from manim import Group, config, logger, tempconfig
from manim.__main__ import main
from manim.renderer.shader import shader_program_cache

from ..constants import RendererType

try:
 from IPython import get_ipython
 from IPython.core.interactiveshell import InteractiveShell
 from IPython.core.magic import (
 Magics,
 line_cell_magic,
 magics_class,
 needs_local_scope,
)
 from IPython.display import Image, Video, display
except ImportError:
 pass
else:

 @magics_class
 class ManimMagic(Magics):
 def __init__(self, shell: InteractiveShell) -> None:
 super().__init__(shell)
 self.rendered_files = {}

 @needs_local_scope
 @line_cell_magic
 def manim(
 self,
 line: str,
 cell: str = None,
 local_ns: dict[str, Any] = None,
) -> None:
 r"""Render Manim scenes contained in IPython cells.
 Works as a line or cell magic.

 .. hint::

 This line and cell magic works best when used in a JupyterLab
 environment: while all of the functionality is available for
 classic Jupyter notebooks as well, it is possible that videos
 sometimes don't update on repeated execution of the same cell
 if the scene name stays the same.

 This problem does not occur when using JupyterLab.

 Please refer to `<https://jupyter.org/>`_ for more information about JupyterLab
 and Jupyter notebooks.

 Usage in line mode::

 %manim [CLI options] MyAwesomeScene

 Usage in cell mode::

 %%manim [CLI options] MyAwesomeScene

 class MyAwesemeScene(Scene):
 def construct(self):
 ...

 Run ``%manim --help`` and ``%manim render --help`` for possible command line interface options.

 .. note::

 The maximal width of the rendered videos that are displayed in the notebook can be
 configured via the ``media_width`` configuration option. The default is set to ``25vw``,
 which is 25% of your current viewport width. To allow the output to become as large
 as possible, set ``config.media_width = "100%"``.

 The ``media_embed`` option will embed the image/video output in the notebook. This is

```

generally undesirable as it makes the notebooks very large, but is required on some platforms (notably Google's CoLab, where it is automatically enabled unless suppressed by ``config.embed = False``) and needed in cases when the notebook (or converted HTML file) will be moved relative to the video locations. Use-cases include building documentation with Sphinx and JupyterBook. See also the :mod:`manim` directive for Sphinx <manim.utils.docbuild.manim\_directive>.

Examples  
-----

First make sure to put ``import manim``, or even ``from manim import \*`` in a cell and evaluate it. Then, a typical Jupyter notebook cell for Manim could look as follows::

```
%%manim -v WARNING --disable_caching -qm BannerExample
```

```
config.media_width = "75%"
config.media_embed = True
```

```
class BannerExample(Scene):
 def construct(self):
 self.camera.background_color = "#ece6e2"
 banner_large = ManimBanner(dark_theme=False).scale(0.7)
 self.play(banner_large.create())
 self.play(banner_large.expand())
```

Evaluating this cell will render and display the ``BannerExample`` scene defined in the body of the cell.

.. note::

In case you want to hide the red box containing the output progress bar, the ``progress\_bar`` config option should be set to ``None``. This can also be done by passing ``--progress\_bar None`` as a CLI flag.

```
"""
```

```
if cell:
 exec(cell, local_ns)
```

```
args = line.split()
```

```
if not len(args) or "-h" in args or "--help" in args or "--version" in args:
 main(args, standalone_mode=False, prog_name="manim")
 return
```

```
modified_args = self.add_additional_args(args)
args = main(modified_args, standalone_mode=False, prog_name="manim")
with tempconfig(local_ns.get("config", {})):
 config.digest_args(args)
```

```
renderer = None
if config.renderer == RendererType.OPENGL:
 from manim.renderer.opengl_renderer import OpenGLRenderer
```

```
renderer = OpenGLRenderer()
```

```
try:
```

```
 SceneClass = local_ns[config["scene_names"][0]]
 scene = SceneClass(renderer=renderer)
 scene.render()
```

```
finally:
```

```
 # Shader cache becomes invalid as the context is destroyed
 shader_program_cache.clear()
```

```
 # Close OpenGL window here instead of waiting for the main thread to
 # finish causing the window to stay open and freeze
 if renderer is not None and renderer.window is not None:
 renderer.window.close()
```

```
if config["output_file"] is None:
 logger.info("No output file produced")
 return
```

```
local_path = Path(config["output_file"]).relative_to(Path.cwd())
tmpfile = (
 Path(config["media_dir"])
 / "jupyter"
 / f"{{_generate_file_name()}}{local_path.suffix}"
)
```

```
if local_path in self.rendered_files:
 self.rendered_files[local_path].unlink()
self.rendered_files[local_path] = tmpfile
tmpfile.parent.mkdir(parents=True, exist_ok=True)
shutil.copy(local_path, tmpfile)
```

```
file_type = mimetypes.guess_type(config["output_file"])[0]
embed = config["media_embed"]
```

```
if embed is None:
 # videos need to be embedded when running in google colab.
 # do this automatically in case config.media_embed has not been
```

```
set explicitly.
embed = "google.colab" in str(get_ipython())

if file_type.startswith("image"):
 result = Image(filename=config["output_file"])
else:
 result = Video(
 tmpfile,
 html_attributes=f'controls autoplay loop style="max-width: {config["media_width"]};"',
 embed=embed,
)

display(result)

def add_additional_args(self, args: list[str]) -> list[str]:
 additional_args = ["--jupyter"]
 # Use webm to support transparency
 if "-t" in args and "--format" not in args:
 additional_args += ["--format", "webm"]
 return additional_args + args[:-1] + [""]
 + [args[-1]]]

def _generate_file_name() -> str:
 return config["scene_names"][0] + "@" + datetime.now().strftime("%Y-%m-%d@%H-%M-%S")
```

# frames\_testers.py

```
from __future__ import annotations

import contextlib
import warnings
from pathlib import Path

import numpy as np

from manim import logger

from ._show_diff import show_diff_helper

FRAME_ABSOLUTE_TOLERANCE = 1.01
FRAME_MISMATCH_RATIO_TOLERANCE = 1e-5

class _FramesTester:
 def __init__(self, file_path: Path, show_diff=False) -> None:
 self._file_path = file_path
 self._show_diff = show_diff
 self._frames: np.ndarray
 self._number_frames: int = 0
 self._frames_compared = 0

 @contextlib.contextmanager
 def testing(self):
 with np.load(self._file_path) as data:
 self._frames = data["frame_data"]
 # For backward compatibility, when the control data contains only one frame (<= v0.8.0)
 if len(self._frames.shape) != 4:
 self._frames = np.expand_dims(self._frames, axis=0)
 logger.debug(self._frames.shape)
 self._number_frames = np.ma.size(self._frames, axis=0)
 yield
 assert self._frames_compared == self._number_frames, (
 f"The scene tested contained {self._frames_compared} frames, "
 f"when there are {self._number_frames} control frames for this test."
)

 def check_frame(self, frame_number: int, frame: np.ndarray):
 assert frame_number < self._number_frames, (
 f"The tested scene is at frame number {frame_number} "
 f"when there are {self._number_frames} control frames."
)
 try:
 np.testing.assert_allclose(
 frame,
 self._frames[frame_number],
 atol=FRAME_ABSOLUTE_TOLERANCE,
 err_msg=f"Frame no {frame_number}. You can use --show_diff to visually show the difference.",
 verbose=False,
)
 self._frames_compared += 1
 except AssertionError as e:
 number_of_matches = np.isclose(
 frame, self._frames[frame_number], atol=FRAME_ABSOLUTE_TOLERANCE
).sum()
 number_of_mismatches = frame.size - number_of_matches
 if number_of_mismatches / frame.size < FRAME_MISMATCH_RATIO_TOLERANCE:
 # we tolerate a small (< 0.001%) amount of pixel value errors
 # in the tests, this accounts for minor OS dependent inconsistencies
 self._frames_compared += 1
 warnings.warn(
 f"Mismatch of {number_of_mismatches} pixel values in frame {frame_number} "
 f"against control data in {self._file_path}. Below error threshold, "
 "continuing..."
)
 return

 if self._show_diff:
 show_diff_helper(
 frame_number,
 frame,
 self._frames[frame_number],
 self._file_path.name,
)
 raise e

class _ControlDataWriter(_FramesTester):
 def __init__(self, file_path: Path, size_frame: tuple) -> None:
```

```
self.file_path = file_path
self.frames = np.empty((0, *size_frame, 4))
self._number_frames_written: int = 0

Actually write a frame.
def check_frame(self, index: int, frame: np.ndarray):
 frame = frame[np.newaxis, ...]
 self.frames = np.concatenate((self.frames, frame))
 self._number_frames_written += 1

@contextlib.contextmanager
def testing(self):
 yield
 self.save_control_data()

def save_control_data(self):
 self.frames = self.frames.astype("uint8")
 np.savez_compressed(self.file_path, frame_data=self.frames)
 logger.info(
 f"{self._number_frames_written} control frames saved in {self.file_path}",
)
```

## camera.py

```
"""A camera converts the mobjects contained in a Scene into an array of pixels."""

from __future__ import annotations
__all__ = ["Camera", "BackgroundColoredVMobjectDisplayer"]

import copy
import itertools as it
import operator as op
import pathlib
import time
from functools import reduce
from typing import Any, Callable, Iterable

import cairo
import numpy as np
from PIL import Image
from scipy.spatial.distance import pdist

from .. import config, logger
from ..constants import *
from ..mobject.mobject import Mobject
from ..mobject.types.image_mobject import AbstractImageMobject
from ..mobject.types.point_cloud_mobject import PMobject
from ..mobject.types.vectorized_mobject import VMobject
from ..utils.color import color_to_int_rgba
from ..utils.family import extract_mobject_family_members
from ..utils.images import get_full_raster_image_path
from ..utils.iterables import list_difference_update
from ..utils.space_ops import angle_of_vector

LINE_JOIN_MAP = {
 LineJoinType.AUTO: None, # TODO: this could be improved
 LineJoinType.ROUND: cairo.LineJoin.ROUND,
 LineJoinType.BEVEL: cairo.LineJoin.BEVEL,
 LineJoinType.MITER: cairo.LineJoin.MITER,
}

class Camera:
 """Base camera class.

 This is the object which takes care of what exactly is displayed
 on screen at any given moment.

 Parameters

 background_image
 The path to an image that should be the background image.
 If not set, the background is filled with :attr:`self.background_color`

 background
 What :attr:`background` is set to. By default, ``None``.

 pixel_height
 The height of the scene in pixels.

 pixel_width
 The width of the scene in pixels.

 kwargs
 Additional arguments (``background_color``, ``background_opacity``)
 to be set.
 """

 def __init__(
 self,
 background_image: str | None = None,
 frame_center: np.ndarray = ORIGIN,
 image_mode: str = "RGBA",
 n_channels: int = 4,
 pixel_array_dtype: str = "uint8",
 cairo_line_width_multiple: float = 0.01,
 use_z_index: bool = True,
 background: np.ndarray | None = None,
 pixel_height: int | None = None,
 pixel_width: int | None = None,
 frame_height: float | None = None,
 frame_width: float | None = None,
 frame_rate: float | None = None,
 **kwargs,
):
 self.background_image = background_image
 self.frame_center = frame_center
 self.image_mode = image_mode
 self.n_channels = n_channels
 self.pixel_array_dtype = pixel_array_dtype
 self.cairo_line_width_multiple = cairo_line_width_multiple
 self.use_z_index = use_z_index
 self.background = background

 if pixel_height is None:
 pixel_height = config["pixel_height"]
 self.pixel_height = pixel_height

 if pixel_width is None:
 pixel_width = config["pixel_width"]
 self.pixel_width = pixel_width

 if frame_height is None:
 frame_height = config["frame_height"]
 self.frame_height = frame_height

 if frame_width is None:
 frame_width = config["frame_width"]
 self.frame_width = frame_width

 if frame_rate is None:
 frame_rate = config["frame_rate"]
 self.frame_rate = frame_rate

 # TODO: change this to not use kwargs.get
 for attr in ["background_color", "background_opacity"]:
 setattr(self, f"_{attr}", kwargs.get(attr, config[attr]))

 # This one is in the same boat as the above, but it doesn't have the
```

```

same name as the corresponding key so it has to be handled on its own
self.max_allowable_norm = config["frame_width"]

self.rgb_max_val = np.iinfo(self.pixel_array_dtype).max
self.pixel_array_to_cairo_context = {}

Contains the correct method to process a list of Mobjects of the
corresponding class. If a Mobject is not an instance of a class in
this dict (or an instance of a class that inherits from a class in
this dict), then it cannot be rendered.

self.init_background()
self.resize_frame_shape()
self.reset()

def __deepcopy__(self, memo):
 # This is to address a strange bug where deepcopying
 # will result in a segfault, which is somehow related
 # to the aggdraw library
 self.canvas = None
 return copy.copy(self)

@property
def background_color(self):
 return self._background_color

@background_color.setter
def background_color(self, color):
 self._background_color = color
 self.init_background()

@property
def background_opacity(self):
 return self._background_opacity

@background_opacity.setter
def background_opacity(self, alpha):
 self._background_opacity = alpha
 self.init_background()

def type_or_raise(mobject: Mobject):
 """Return the type of mobject, if it is a type that can be rendered.

 If `mobject` is an instance of a class that inherits from a class that
 can be rendered, return the super class. For example, an instance of a
 Square is also an instance of VMobject, and these can be rendered.
 Therefore, `type_or_raise(Square())` returns True.

 Parameters

 mobject
 The object to take the type of.

 Notes

 For a list of classes that can currently be rendered, see :meth:`display_funcs`.

 Returns

 Type[:class:`~.Mobject`]
 The type of mobjects, if it can be rendered.

 Raises

 :exc:`TypeError`
 When mobject is not an instance of a class that can be rendered.
 """
 self.display_funcs = {
 VMobject: self.display_multiple_vectorized_mobjects,
 PMobject: self.display_multiple_point_cloud_mobjects,
 AbstractImageMobject: self.display_multiple_image_mobjects,
 Mobject: lambda batch, pa: batch, # Do nothing
 }
 # We have to check each type in turn because we are dealing with
 # super classes. For example, if square = Square(), then
 # type(square) != VMobject, but isinstance(square, VMobject) == True.
 for _type in self.display_funcs:
 if isinstance(mobject, _type):
 return _type
 raise TypeError(f"Displaying an object of class {_type} is not supported")

def reset_pixel_shape(self, new_height: float, new_width: float):
 """This method resets the height and width
 of a single pixel to the passed new_height and new_width.

 Parameters

 new_height
 The new height of the entire scene in pixels
 new_width
 The new width of the entire scene in pixels
 """
 self.pixel_width = new_width
 self.pixel_height = new_height
 self.init_background()
 self.resize_frame_shape()
 self.reset()

def resize_frame_shape(self, fixed_dimension: int = 0):
 """
 Changes frame_shape to match the aspect ratio
 of the pixels, where fixed_dimension determines
 whether frame_height or frame_width
 remains fixed while the other changes accordingly.

 Parameters

 fixed_dimension
 If 0, height is scaled with respect to width
 else, width is scaled with respect to height.
 """
 pixel_height = self.pixel_height
 pixel_width = self.pixel_width
 frame_height = self.frame_height
 frame_width = self.frame_width
 aspect_ratio = pixel_width / pixel_height
 if fixed_dimension == 0:
 frame_height = frame_width / aspect_ratio
 else:

```

```

 frame_width = aspect_ratio * frame_height
 self.frame_height = frame_height
 self.frame_width = frame_width

 def init_background(self):
 """Initialize the background.
 If self.background_image is the path of an image
 the image is set as background; else, the default
 background color fills the background.
 """
 height = self.pixel_height
 width = self.pixel_width
 if self.background_image is not None:
 path = get_full_raster_image_path(self.background_image)
 image = Image.open(path).convert(self.image_mode)
 # TODO, how to gracefully handle backgrounds
 # with different sizes?
 self.background = np.array(image)[:height, :width]
 self.background = self.background.astype(self.pixel_array_dtype)
 else:
 background_rgba = color_to_int_rgba(
 self.background_color,
 self.background_opacity,
)
 self.background = np.zeros(
 (height, width, self.n_channels),
 dtype=self.pixel_array_dtype,
)
 self.background[:, :] = background_rgba

 def get_image(self, pixel_array: np.ndarray | list | tuple | None = None):
 """Returns an image from the passed
 pixel array, or from the current frame
 if the passed pixel array is none.

 Parameters

 pixel_array
 The pixel array from which to get an image, by default None

 Returns

 PIL.Image
 The PIL image of the array.
 """
 if pixel_array is None:
 pixel_array = self.pixel_array
 return Image.fromarray(pixel_array, mode=self.image_mode)

 def convert_pixel_array(
 self, pixel_array: np.ndarray | list | tuple, convert_from_floats: bool = False
):
 """Converts a pixel array from values that have floats in them
 to proper RGB values.

 Parameters

 pixel_array
 Pixel array to convert.
 convert_from_floats
 Whether or not to convert float values to ints, by default False

 Returns

 np.array
 The new, converted pixel array.
 """
 retval = np.array(pixel_array)
 if convert_from_floats:
 retval = np.apply_along_axis(
 lambda f: (f * self.rgb_max_val).astype(self.pixel_array_dtype),
 2,
 retval,
)
 return retval

 def set_pixel_array(
 self, pixel_array: np.ndarray | list | tuple, convert_from_floats: bool = False
):
 """Sets the pixel array of the camera to the passed pixel array.

 Parameters

 pixel_array
 The pixel array to convert and then set as the camera's pixel array.
 convert_from_floats
 Whether or not to convert float values to proper RGB values, by default False
 """
 converted_array = self.convert_pixel_array(pixel_array, convert_from_floats)
 if not (
 hasattr(self, "pixel_array")
 and self.pixel_array.shape == converted_array.shape
):
 self.pixel_array = converted_array
 else:
 # Set in place
 self.pixel_array[:, :, :] = converted_array[:, :, :]

 def set_background(
 self, pixel_array: np.ndarray | list | tuple, convert_from_floats: bool = False
):
 """Sets the background to the passed pixel_array after converting
 to valid RGB values.

 Parameters

 pixel_array
 The pixel array to set the background to.
 convert_from_floats
 Whether or not to convert float values to proper RGB valid ones, by default False
 """
 self.background = self.convert_pixel_array(pixel_array, convert_from_floats)

 # TODO, this should live in utils, not as a method of Camera
 def make_background_from_func(
 self, coords_to_colors_func: Callable[[np.ndarray], np.ndarray]
):
 """Makes a pixel array for the background by using coords_to_colors_func to determine each pixel's color. Each input

```

pixel's color. Each input to coords\_to\_colors\_func is an (x, y) pair in space (in ordinary space coordinates; not pixel coordinates), and each output is expected to be an RGBA array of 4 floats.

**Parameters**

-----

coords\_to\_colors\_func  
The function whose input is an (x,y) pair of coordinates and  
whose return values must be the colors for that point

**Returns**

-----

np.array  
The pixel array which can then be passed to set\_background.

"""

```
logger.info("Starting set_background")
coords = self.get_coords_of_all_pixels()
new_background = np.apply_along_axis(coords_to_colors_func, 2, coords)
logger.info("Ending set_background")

return self.convert_pixel_array(new_background, convert_from_floats=True)
```

def set\_background\_from\_func(  
self, coords\_to\_colors\_func: Callable[[np.ndarray], np.ndarray]  
):  
"""
Sets the background to a pixel array using coords\_to\_colors\_func to determine each pixel's color. Each input  
pixel's color. Each input to coords\_to\_colors\_func is an (x, y) pair in space (in ordinary space coordinates; not  
pixel coordinates), and each output is expected to be an RGBA array of 4 floats.

**Parameters**

-----

coords\_to\_colors\_func  
The function whose input is an (x,y) pair of coordinates and  
whose return values must be the colors for that point

"""

```
self.set_background(self.make_background_from_func(coords_to_colors_func))
```

def reset(self):  
"""Resets the camera's pixel array  
to that of the background

**Returns**

-----

Camera  
The camera object after setting the pixel array.

"""

```
self.set_pixel_array(self.background)
return self
```

def set\_frame\_to\_background(self, background):  
self.set\_pixel\_array(background)

####

def get\_mobjects\_to\_display(  
self,  
mobjects: Iterable[Mobject],  
include\_submobjects: bool = True,  
excluded\_mobjects: list | None = None,  
):  
"""Used to get the list of mobjects to display  
with the camera.

**Parameters**

-----

mobjects  
The Mobjects  
include\_submobjects  
Whether or not to include the submobjects of mobjects, by default True  
excluded\_mobjects  
Any mobjects to exclude, by default None

**Returns**

-----

list  
list of mobjects

"""

```
if include_submobjects:

mobjects = extract_mobject_family_members(

mobjects,

use_z_index=self.use_z_index,

only_those_with_points=True,

)

if excluded_mobjects:

all_excluded = extract_mobject_family_members(

excluded_mobjects,

use_z_index=self.use_z_index,

)

mobjects = list_difference_update(mobjects, all_excluded)
return list(mobjects)
```

def is\_in\_frame(self, mobject: Mobject):  
"""Checks whether the passed mobject is in  
frame or not.

**Parameters**

-----

mobject  
The mobject for which the checking needs to be done.

**Returns**

-----

bool  
True if in frame, False otherwise.

"""

```
fc = self.frame_center
fh = self.frame_height
fw = self.frame_width
return not reduce(
op.or_,

[
 mobject.get_right()[0] < fc[0] - fw / 2,
 mobject.get_bottom()[1] > fc[1] + fh / 2,
 mobject.get_left()[0] > fc[0] + fw / 2,
 mobject.get_top()[1] < fc[1] - fh / 2,
],
),
```

```

def capture_mobject(self, mobject: Mobject, **kwargs: Any):
 """Capture mobjects by storing it in :attr:`pixel_array`.

This is a single-mobject version of :meth:`capture_mobjects`.

Parameters

mobject
 Mobject to capture.

kwargs
 Keyword arguments to be passed to :meth:`get_mobjects_to_display`.

"""
 return self.capture_mobjects([mobject], **kwargs)

def capture_mobjects(self, mobjects: Iterable[Mobject], **kwargs: Any):
 """Capture mobjects by printing them on :attr:`pixel_array`.

This is the essential function that converts the contents of a Scene
into an array, which is then converted to an image or video.

Parameters

mobjects
 Mobjects to capture.

kwargs
 Keyword arguments to be passed to :meth:`get_mobjects_to_display`.

Notes

For a list of classes that can currently be rendered, see :meth:`display_funcs`.

"""
 # The mobjects will be processed in batches (or runs) of mobjects of
the same type. That is, if the list mobjects contains objects of
types [VMobject, VMobject, VMobject, PMobject, VMobject],
then they will be captured in three batches: [VMobject, VMobject,
VMobject], [PMobject, PMobject], and [VMobject]. This must be done
without altering their order. it.groupby computes exactly this
partition while at the same time preserving order.
mobjects = self.get_mobjects_to_display(mobjects, **kwargs)
for group_type, group in it.groupby(mobjects, self.type_or_raise):
 self.display_funcs[group_type](list(group), self.pixel_array)

Methods associated with svg rendering

NOTE: None of the methods below have been mentioned outside of their definitions. Their DocStrings are not as
detailed as possible.

def get_cached_cairo_context(self, pixel_array: np.ndarray):
 """Returns the cached cairo context of the passed
pixel array if it exists, and None if it doesn't.

Parameters

pixel_array
 The pixel array to check.

Returns

cairo.Context
 The cached cairo context.

"""
 return self.pixel_array_to_cairo_context.get(id(pixel_array), None)

def cache_cairo_context(self, pixel_array: np.ndarray, ctx: cairo.Context):
 """Caches the passed Pixel array into a Cairo Context

Parameters

pixel_array
 The pixel array to cache
ctx
 The context to cache it into.

"""
 self.pixel_array_to_cairo_context[id(pixel_array)] = ctx

def get_cairo_context(self, pixel_array: np.ndarray):
 """Returns the cairo context for a pixel array after
caching it to self.pixel_array_to_cairo_context
If that array has already been cached, it returns the
cached version instead.

Parameters

pixel_array
 The Pixel array to get the cairo context of.

Returns

cairo.Context
 The cairo context of the pixel array.

"""
 cached_ctx = self.get_cached_cairo_context(pixel_array)
 if cached_ctx:
 return cached_ctx
 pw = self.pixel_width
 ph = self.pixel_height
 fw = self.frame_width
 fh = self.frame_height
 fc = self.frame_center
 surface = cairo.ImageSurface.create_for_data(
 pixel_array,
 cairo.FORMAT_ARGB32,
 pw,
 ph,
)
 ctx = cairo.Context(surface)
 ctx.scale(pw, ph)
 ctx.set_matrix(
 cairo.Matrix(
 (pw / fw),
 0,
 0,
 -(ph / fh),
 (pw / 2) - fc[0] * (pw / fw),
 (ph / 2) + fc[1] * (ph / fh),
)
)

```

```

),
 self.cache_cairo_context(pixel_array, ctx)
 return ctx

def display_multiple_vectorized_vmobjects(
 self, vmobjects: list, pixel_array: np.ndarray
):
 """Displays multiple VMobjects in the pixel_array

 Parameters

 vmobjects
 list of VMobjects to display
 pixel_array
 The pixel array
 """
 if len(vmobjects) == 0:
 return
 batch_image_pairs = it.groupby(vmobjects, lambda vm: vm.get_background_image())
 for image, batch in batch_image_pairs:
 if image:
 self.display_multiple_background_colored_vmobjects(batch, pixel_array)
 else:
 self.display_multiple_non_background_colored_vmobjects(
 batch,
 pixel_array,
)

def display_multiple_non_background_colored_vmobjects(
 self, vmobjects: list, pixel_array: np.ndarray
):
 """Displays multiple VMobjects in the cairo context, as long as they don't have
 background colors.

 Parameters

 vmobjects
 list of the VMobjects
 pixel_array
 The Pixel array to add the VMobjects to.
 """
 ctx = self.get_cairo_context(pixel_array)
 for vmobject in vmobjects:
 self.display_vectorized(vmobject, ctx)

def display_vectorized(self, vmobject: VMobject, ctx: cairo.Context):
 """Displays a VMobject in the cairo context

 Parameters

 vmobject
 The Vectorized Mobject to display
 ctx
 The cairo context to use.

 Returns

 Camera
 The camera object
 """
 self.set_cairo_context_path(ctx, vmobject)
 self.apply_stroke(ctx, vmobject, background=True)
 self.apply_fill(ctx, vmobject)
 self.apply_stroke(ctx, vmobject)
 return self

def set_cairo_context_path(self, ctx: cairo.Context, vmobject: VMobject):
 """Sets a path for the cairo context with the vmobject passed

 Parameters

 ctx
 The cairo context
 vmobject
 The VMobject

 Returns

 Camera
 Camera object after setting cairo_context_path
 """
 points = self.transform_points_pre_display(vmobject, vmobject.points)
 # TODO, shouldn't this be handled in transform_points_pre_display?
 # points = points - self.get_frame_center()
 if len(points) == 0:
 return

 ctx.new_path()
 subpaths = vmobject.gen_subpaths_from_points_2d(points)
 for subpath in subpaths:
 quads = vmobject.gen_cubic_bezier_tuples_from_points(subpath)
 ctx.new_sub_path()
 start = subpath[0]
 ctx.move_to(*start[:2])
 for p0, p1, p2, p3 in quads:
 ctx.curve_to(*p1[:2], *p2[:2], *p3[:2])
 if vmobject.consider_points_equals_2d(subpath[0], subpath[-1]):
 ctx.close_path()
 return self

def set_cairo_context_color(
 self, ctx: cairo.Context, rgbs: np.ndarray, vmobject: VMobject
):
 """Sets the color of the cairo context

 Parameters

 ctx
 The cairo context
 rgbs
 The RGBA array with which to color the context.
 vmobject
 The VMobject with which to set the color.

 Returns

 Camera
 The camera object
 """

```

```

"""
if len(rgbas) == 1:
 # Use reversed rgb because cairo surface is
 # encodes it in reverse order
 ctx.set_source_rgba(*rgbas[0][2::-1], rbgas[0][3])
else:
 points = vmobject.get_gradient_start_and_end_points()
 points = self.transform_points_pre_display(vmobject, points)
 pat = cairo.LinearGradient(*it.chain(*[point[:2] for point in points]))
 step = 1.0 / (len(rgbas) - 1)
 offsets = np.arange(0, 1 + step, step)
 for rgba, offset in zip(rgbas, offsets):
 pat.add_color_stop_rgba(offset, *rgba[2::-1], rgba[3])
 ctx.set_source(pat)
return self

def apply_fill(self, ctx: cairo.Context, vmobject: VMobject):
 """Fills the cairo context

 Parameters

 ctx
 The cairo context
 vmobject
 The VMobject

 Returns

 Camera
 The camera object.

 """
 self.set_cairo_context_color(ctx, self.get_fill_rgbas(vmobject), vmobject)
 ctx.fill_preserve()
 return self

def apply_stroke(
 self, ctx: cairo.Context, vmobject: VMobject, background: bool = False
):
 """Applies a stroke to the VMobject in the cairo context.

 Parameters

 ctx
 The cairo context
 vmobject
 The VMobject
 background
 Whether or not to consider the background when applying this
 stroke width, by default False

 Returns

 Camera
 The camera object with the stroke applied.

 """
 width = vmobject.get_stroke_width(background)
 if width == 0:
 return self
 self.set_cairo_context_color(
 ctx,
 self.get_stroke_rgbas(vmobject, background=background),
 vmobject,
)
 ctx.set_line_width(
 width
 * self.cairo_line_width_multiple
 # This ensures lines have constant width as you zoom in on them.
 * (self.frame_width / self.frame_width),
)
 if vmobject.joint_type != LineJointType.AUTO:
 ctx.set_line_join(LINE_JOIN_MAP[vmobject.joint_type])
 ctx.stroke_preserve()
 return self

def get_stroke_rgbas(self, vmobject: VMobject, background: bool = False):
 """Gets the RGBA array for the stroke of the passed
 VMobject.

 Parameters

 vmobject
 The VMobject
 background
 Whether or not to consider the background when getting the stroke
 RGBAs, by default False

 Returns

 np.ndarray
 The RGBA array of the stroke.

 """
 return vmobject.get_stroke_rgbas(background)

def get_fill_rgbas(self, vmobject: VMobject):
 """Returns the RGBA array of the fill of the passed VMobject

 Parameters

 vmobject
 The VMobject

 Returns

 np.array
 The RGBA Array of the fill of the VMobject

 """
 return vmobject.get_fill_rgbas()

def get_background_colored_vmobject_displayer(self):
 """Returns the background_colored_vmobject_displayer
 if it exists or makes one and returns it if not.

 Returns

 BackGroundColoredVMobjectDisplayer
 Object that displays VMobjects that have the same color
 as the background.

 """
 # Quite wordy to type out a bunch

```

```

bcvd = "background_colored_vmobject_displayer"
if not hasattr(self, bcvd):
 setattr(self, bcvd, BackgroundColoredVMobjectDisplayer(self))
return getattr(self, bcvd)

def display_multiple_background_colored_vmobjects(
 self, cvmobjects: list, pixel_array: np.ndarray
):
 """Displays multiple vmobjects that have the same color as the background.

 Parameters

 cvmobjects
 List of Colored VMobjects
 pixel_array
 The pixel array.

 Returns

 Camera
 The camera object.
 """
 display = self.get_background_colored_vmobject_displayer()
 cvmobject_pixel_array = display(*cvmobjects)
 self.overlay_rgba_array(pixel_array, cvmobject_pixel_array)
 return self

Methods for other rendering

NOTE: Out of the following methods, only `transform_points_pre_display` and `points_to_pixel_coords` have been mentioned outside of their definitions.
As a result, the other methods do not have as detailed docstrings as would be preferred.

def display_multiple_point_cloud_mobjects(
 self, pmobjects: list, pixel_array: np.ndarray
):
 """Displays multiple PMobjects by modifying the passed pixel array.

 Parameters

 pmobjects
 List of PMobjects
 pixel_array
 The pixel array to modify.
 """
 for pmobject in pmobjects:
 self.display_point_cloud(
 pmobject,
 pmobject.points,
 pmobject.rgbas,
 self.adjusted_thickness(pmobject.stroke_width),
 pixel_array,
)

def display_point_cloud(
 self,
 pmobject: PMobject,
 points: list,
 rgbas: np.ndarray,
 thickness: float,
 pixel_array: np.ndarray,
):
 """Displays a PMobject by modifying the pixel array suitably.

 TODO: Write a description for the rgbas argument.

 Parameters

 pmobject
 Point Cloud Mobject
 points
 The points to display in the point cloud mobject
 rgbas
 The thickness of each point of the PMobject
 pixel_array
 The pixel array to modify.

 """
 if len(points) == 0:
 return
 pixel_coords = self.points_to_pixel_coords(pmobject, points)
 pixel_coords = self.thickened_coordinates(pixel_coords, thickness)
 rgba_len = pixel_array.shape[2]
 rgbas = (self.rgb_max_val * rgbas).astype(self.pixel_array.dtype)
 target_len = len(pixel_coords)
 factor = target_len // len(rgbas)
 rgbas = np.array([rgbas] * factor).reshape((target_len, rgba_len))

 on_screen_indices = self.on_screen_pixels(pixel_coords)
 pixel_coords = pixel_coords[on_screen_indices]
 rgbas = rgbas[on_screen_indices]

 ph = self.pixel_height
 pw = self.pixel_width

 flattener = np.array([1, pw], dtype="int")
 flattener = flattener.reshape((2, 1))
 indices = np.dot(pixel_coords, flattener)[:, 0]
 indices = indices.astype("int")

 new_pa = pixel_array.reshape((ph * pw, rgba_len))
 new_pa[indices] = rgbas
 pixel_array[:, :] = new_pa.reshape((ph, pw, rgba_len))

def display_multiple_image_mobjects(
 self, image_mobjects: list, pixel_array: np.ndarray
):
 """Displays multiple image mobjects by modifying the passed pixel_array.

 Parameters

 image_mobjects
 list of ImageMobjects
 pixel_array
 The pixel array to modify.
 """
 for image_mobject in image_mobjects:

```

```

 self.display_image_mobject(image_mobject, pixel_array)

def display_image_mobject(
 self, image_mobject: AbstractImageMobject, pixel_array: np.ndarray
):
 """Displays an ImageMobject by changing the pixel_array suitably.

 Parameters

 image_mobject
 The imageMobject to display
 pixel_array
 The Pixel array to put the imagemobject in.

 """
 corner_coords = self.points_to_pixel_coords(image_mobject, image_mobject.points)
 ul_coords, ur_coords, dl_coords = corner_coords
 right_vect = ur_coords - ul_coords
 down_vect = dl_coords - ul_coords
 center_coords = ul_coords + (right_vect + down_vect) / 2

 sub_image = Image.fromarray(image_mobject.get_pixel_array(), mode="RGBA")

 # Reshape
 pixel_width = max(int(pdist([ul_coords, ur_coords])), 1)
 pixel_height = max(int(pdist([ul_coords, dl_coords])), 1)
 sub_image = sub_image.resize(
 (pixel_width, pixel_height),
 resample=image_mobject.resampling_algorithm,
)

 # Rotate
 angle = angle_of_vector(right_vect)
 adjusted_angle = -int(360 * angle / TAU)
 if adjusted_angle != 0:
 sub_image = sub_image.rotate(
 adjusted_angle,
 resample=image_mobject.resampling_algorithm,
 expand=1,
)

 # TODO, there is no accounting for a shear...

 # Paste into an image as large as the camera's pixel array
 full_image = Image.fromarray(
 np.zeros((self.pixel_height, self.pixel_width)),
 mode="RGBA",
)
 new_ul_coords = center_coords - np.array(sub_image.size) / 2
 new_ul_coords = new_ul_coords.astype(int)
 full_image.paste(
 sub_image,
 box=(
 new_ul_coords[0],
 new_ul_coords[1],
 new_ul_coords[0] + sub_image.size[0],
 new_ul_coords[1] + sub_image.size[1],
),
)
 # Paint on top of existing pixel array
 self.overlay_PIL_image(pixel_array, full_image)

def overlay_rgba_array(self, pixel_array: np.ndarray, new_array: np.ndarray):
 """Overlays an RGBA array on top of the given Pixel array.

 Parameters

 pixel_array
 The original pixel array to modify.
 new_array
 The new pixel array to overlay.

 """
 self.overlay_PIL_image(pixel_array, self.get_image(new_array))

def overlay_PIL_image(self, pixel_array: np.ndarray, image: Image):
 """Overlays a PIL image on the passed pixel array.

 Parameters

 pixel_array
 The Pixel array
 image
 The Image to overlay.

 """
 pixel_array[:, :] = np.array(
 Image.alpha_composite(self.get_image(pixel_array), image),
 dtype="uint8",
)

def adjust_out_of_range_points(self, points: np.ndarray):
 """If any of the points in the passed array are out of the viable range, they are adjusted suitably.

 Parameters

 points
 The points to adjust

 Returns

 np.array
 The adjusted points.

 """
 if not np.any(points > self.max_allowable_norm):
 return points
 norms = np.apply_along_axis(np.linalg.norm, 1, points)
 violator_indices = norms > self.max_allowable_norm
 violators = points[violator_indices, :]
 violator_norms = norms[violator_indices]
 reshaped_norms = np.repeat(
 violator_norms.reshape((len(violator_norms), 1)),
 points.shape[1],
 1,
)
 rescaled = self.max_allowable_norm * violators / reshaped_norms
 points[violator_indices] = rescaled
 return points

def transform_points_pre_display(
 self,

```

```

mobject,
points,
): # TODO: Write more detailed docstrings for this method.
NOTE: There seems to be an unused argument `mobject`.

 # Subclasses (like ThreeDCamera) may want to
 # adjust points further before they're shown
 if not np.all(np.isfinite(points)):
 # TODO, print some kind of warning about
 # mobject having invalid points?
 points = np.zeros((1, 3))
 return points

def points_to_pixel_coords(
 self,
 mobject,
 points,
): # TODO: Write more detailed docstrings for this method.
 points = self.transform_points_pre_display(mobject, points)
 shifted_points = points - self.frame_center

 result = np.zeros((len(points), 2))
 pixel_height = self.pixel_height
 pixel_width = self.pixel_width
 frame_height = self.frame_height
 frame_width = self.frame_width
 width_mult = pixel_width / frame_width
 width_add = pixel_width / 2
 height_mult = pixel_height / frame_height
 height_add = pixel_height / 2
 # Flip on y-axis as you go
 height_mult *= -1

 result[:, 0] = shifted_points[:, 0] * width_mult + width_add
 result[:, 1] = shifted_points[:, 1] * height_mult + height_add
 return result.astype("int")

def on_screen_pixels(self, pixel_coords: np.ndarray):
 """Returns array of pixels that are on the screen from a given
 array of pixel_coordinates

 Parameters

 pixel_coords
 The pixel coords to check.

 Returns

 np.array
 The pixel coords on screen.
 """
 return reduce(
 op.and_,
 [
 pixel_coords[:, 0] >= 0,
 pixel_coords[:, 0] < self.pixel_width,
 pixel_coords[:, 1] >= 0,
 pixel_coords[:, 1] < self.pixel_height,
],
)

def adjusted_thickness(self, thickness: float) -> float:
 """Computes the adjusted stroke width for a zoomed camera.

 Parameters

 thickness
 The stroke width of a mobject.

 Returns

 float
 The adjusted stroke width that reflects zooming in with
 the camera.
 """
 # TODO: This seems...unsystematic
 big_sum = op.add(config["pixel_height"], config["pixel_width"])
 this_sum = op.add(self.pixel_height, self.pixel_width)
 factor = big_sum / this_sum
 return 1 + (thickness - 1) * factor

def get_thickening_nudges(self, thickness: float):
 """Determine a list of vectors used to nudge
 two-dimensional pixel coordinates.

 Parameters

 thickness

 Returns

 np.array

 """
 thickness = int(thickness)
 _range = list(range(-thickness // 2 + 1, thickness // 2 + 1))
 return np.array(list(it.product(_range, _range)))

def thickened_coordinates(self, pixel_coords: np.ndarray, thickness: float):
 """Returns thickened coordinates for a passed array of pixel coords and
 a thickness to thicken by.

 Parameters

 pixel_coords
 Pixel coordinates
 thickness
 Thickness

 Returns

 np.array
 Array of thickened pixel coords.
 """
 nudges = self.get_thickening_nudges(thickness)
 pixel_coords = np.array([pixel_coords + nudge for nudge in nudges])
 size = pixel_coords.size
 return pixel_coords.reshape((size // 2, 2))

```

```

TODO, reimplement using cairo matrix
def get_coords_of_all_pixels(self):
 """Returns the cartesian coordinates of each pixel.

 Returns

 np.ndarray
 The array of cartesian coordinates.
 """
 # These are in x, y order, to help me keep things straight
 full_space_dims = np.array([self.frame_width, self.frame_height])
 full_pixel_dims = np.array([self.pixel_width, self.pixel_height])

 # These are addressed in the same y, x order as in pixel_array, but the values in them
 # are listed in x, y order
 uncentered_pixel_coords = np.indices([self.pixel_height, self.pixel_width])[::-1]
 .transpose(1, 2, 0)
 uncentered_space_coords = (
 uncentered_pixel_coords * full_space_dims
) / full_pixel_dims
 # Could structure above line's computation slightly differently, but figured (without much
 # thought) multiplying by frame_shape first, THEN dividing by pixel_shape, is probably
 # better than the other order, for avoiding underflow quantization in the division (whereas
 # overflow is unlikely to be a problem)

 centered_space_coords = uncentered_space_coords - (full_space_dims / 2)

 # Have to also flip the y coordinates to account for pixel array being listed in
 # top-to-bottom order, opposite of screen coordinate convention
 centered_space_coords = centered_space_coords * (1, -1)

 return centered_space_coords

NOTE: The methods of the following class have not been mentioned outside of their definitions.
Their DocStrings are not as detailed as preferred.
class BackgroundColoredVMOBJobjectDisplayer:
 """Auxiliary class that handles displaying vectorized mobjects with
 a set background image.

 Parameters

 camera
 Camera object to use.
 """

 def __init__(self, camera: Camera):
 self.camera = camera
 self.file_name_to_pixel_array_map = {}
 self.pixel_array = np.array(camera.pixel_array)
 self.reset_pixel_array()

 def reset_pixel_array(self):
 self.pixel_array[:, :] = 0

 def resize_background_array(
 self,
 background_array: np.ndarray,
 new_width: float,
 new_height: float,
 mode: str = "RGBA",
):
 """Resizes the pixel array representing the background.

 Parameters

 background_array
 The pixel
 new_width
 The new width of the background
 new_height
 The new height of the background
 mode
 The PIL image mode, by default "RGBA"

 Returns

 np.array
 The numpy pixel array of the resized background.
 """
 image = Image.fromarray(background_array)
 image = image.convert(mode)
 resized_image = image.resize((new_width, new_height))
 return np.array(resized_image)

 def resize_background_array_to_match(
 self, background_array: np.ndarray, pixel_array: np.ndarray
):
 """Resizes the background array to match the passed pixel array.

 Parameters

 background_array
 The prospective pixel array.
 pixel_array
 The pixel array whose width and height should be matched.

 Returns

 np.array
 The resized background array.
 """
 height, width = pixel_array.shape[:2]
 mode = "RGBA" if pixel_array.shape[2] == 4 else "RGB"
 return self.resize_background_array(background_array, width, height, mode)

 def get_background_array(self, image: Image.Image | pathlib.Path | str):
 """Gets the background array that has the passed file_name.

 Parameters

 image
 The background image or its file name.

 Returns

 np.ndarray
 The pixel array of the image.

```

```

"""
image_key = str(image)

if image_key in self.file_name_to_pixel_array_map:
 return self.file_name_to_pixel_array_map[image_key]
if isinstance(image, str):
 full_path = get_full_raster_image_path(image)
 image = Image.open(full_path)
 back_array = np.array(image)

pixel_array = self.pixel_array
if not np.all(pixel_array.shape == back_array.shape):
 back_array = self.resize_background_array_to_match(back_array, pixel_array)

self.file_name_to_pixel_array_map[image_key] = back_array
return back_array

def display(self, *cvmobjects: VMobject):
 """Displays the colored VMobjects.

 Parameters

 *cvmobjects
 The VMobjects

 Returns

 np.array
 The pixel array with the `cvmobjects` displayed.

 """
batch_image_pairs = it.groupby(cvmobjects, lambda cv: cv.get_background_image())
curr_array = None
for image, batch in batch_image_pairs:
 background_array = self.get_background_array(image)
 pixel_array = self.pixel_array
 self.camera.display_multiple_non_background_colored_vmoibjcts(
 batch,
 pixel_array,
)
 new_array = np.array(
 (background_array * pixel_array.astype("float") / 255),
 dtype=self.camera.pixel_array_dtype,
)
 if curr_array is None:
 curr_array = new_array
 else:
 curr_array = np.maximum(curr_array, new_array)
 self.reset_pixel_array()
return curr_array

```

## gui.py

```
dpg.start_dearpygui()
```

# numbers.py

```
"""Animations for changing numbers."""

from __future__ import annotations

__all__ = ["ChangingDecimal", "ChangeDecimalToValue"]

import typing

from manim.mobject.text.numbers import DecimalNumber

from ..animation.animation import Animation
from ..utils.bezier import interpolate

class ChangingDecimal(Animation):
 def __init__(
 self,
 decimal_mob: DecimalNumber,
 number_update_func: typing.Callable[[float], float],
 suspend_mobject_updating: bool | None = False,
 **kwargs,
) -> None:
 self.check_validity_of_input(decimal_mob)
 self.number_update_func = number_update_func
 super().__init__(
 decimal_mob, suspend_mobject_updating=suspend_mobject_updating, **kwargs
)

 def check_validity_of_input(self, decimal_mob: DecimalNumber) -> None:
 if not isinstance(decimal_mob, DecimalNumber):
 raise TypeError("ChangingDecimal can only take in a DecimalNumber")

 def interpolate_mobject(self, alpha: float) -> None:
 self.mobject.set_value(self.number_update_func(self.rate_func(alpha)))

class ChangeDecimalToValue(ChangingDecimal):
 def __init__(
 self,
 decimal_mob: DecimalNumber,
 target_number: int,
 **kwargs
) -> None:
 start_number = decimal_mob.number
 super().__init__(
 decimal_mob, lambda a: interpolate(start_number, target_number, a), **kwargs
)
```

# indication.py

```
"""Animations drawing attention to particular mobjects.

Examples

.. manim:: Indications

class Indications(Scene):
 def construct(self):
 indications = [ApplyWave,Circumscribe,Flash,FocusOn,Indicate>ShowPassingFlash,Wiggle]
 names = [Tex(i.__name__).scale(3) for i in indications]

 self.add(names[0])
 for i in range(len(names)):
 if indications[i] is Flash:
 self.play(Flash(UP))
 elif indications[i] is ShowPassingFlash:
 self.play(ShowPassingFlash(Underline(names[i])))
 else:
 self.play(indications[i](names[i]))
 self.play(AnimationGroup(
 FadeOut(names[i], shift=UP*1.5),
 FadeIn(names[(i+1)%len(names)], shift=UP*1.5),
))
"""

__all__ = [
 "FocusOn",
 "Indicate",
 "Flash",
 "ShowPassingFlash",
 "ShowPassingFlashWithThinningStrokeWidth",
 "ShowCreationThenFadeOut",
 "ApplyWave",
 "Circumscribe",
 "Wiggle",
]
from typing import Callable, Iterable, Optional, Tuple, Type, Union

import numpy as np
from colour import Color

from manim.mobject.geometry.arc import Circle, Dot
from manim.mobject.geometry.line import Line
from manim.mobject.geometry.polygram import Rectangle
from manim.mobject.geometry.shape_matchers import SurroundingRectangle

from .. import config
from ..animation.animation import Animation
from ..animation.composition import AnimationGroup, Succession
from ..animation.creation import Create, ShowPartial, Uncreate
from ..animation.fading import FadeIn, FadeOut
from ..animation.movement import Homotopy
from ..animation.transform import Transform
from ..constants import *
from ..mobject.mobject import Mobject
from ..mobject.types.vectorized_mobject import VGroup, VMobject
from ..utils.bezier import interpolate, inverse_interpolate
from ..utils.color import GREY, YELLOW
from ..utils.deprecation import deprecated
from ..utils.rate_functions import smooth, there_and_back, wiggle
from ..utils.space_ops import normalize

class FocusOn(Transform):
 """Shrink a spotlight to a position.

Parameters

focus_point
 The point at which to shrink the spotlight. If it is a :class:`.~Mobject` its center will be used.
opacity
 The opacity of the spotlight.
color
 The color of the spotlight.
run_time
 The duration of the animation.
kwargs
 Additional arguments to be passed to the :class:`.~Succession` constructor
"""
 pass
```

```

Examples

.. manim:: UsingFocusOn

 class UsingFocusOn(Scene):
 def construct(self):
 dot = Dot(color=YELLOW).shift(DOWN)
 self.add(Tex("Focusing on the dot below:"), dot)
 self.play(FocusOn(dot))
 self.wait()

 """
 def __init__(
 self,
 focus_point: Union[np.ndarray, Mobject],
 opacity: float = 0.2,
 color: str = GREY,
 run_time: float = 2,
 **kwargs
) -> None:
 self.focus_point = focus_point
 self.color = color
 self.opacity = opacity
 remover = True
 starting_dot = Dot(
 radius=config["frame_x_radius"] + config["frame_y_radius"],
 stroke_width=0,
 fill_color=self.color,
 fill_opacity=0,
)
 super().__init__(starting_dot, run_time=run_time, remover=remover, **kwargs)

 def create_target(self) -> Dot:
 little_dot = Dot(radius=0)
 little_dot.set_fill(self.color, opacity=self.opacity)
 little_dot.add_updater(lambda d: d.move_to(self.focus_point))
 return little_dot

class Indicate(Transform):
 """Indicate a Mobject by temporarily resizing and recoloring it.

Parameters

mobject
 The mobject to indicate.
scale_factor
 The factor by which the mobject will be temporally scaled
color
 The color the mobject temporally takes.
rate_func
 The function definig the animation progress at every point in time.
kwargs
 Additional arguments to be passed to the :class:`~.Succession` constructor

Examples

.. manim:: UsingIndicate

 class UsingIndicate(Scene):
 def construct(self):
 tex = Tex("Indicate").scale(3)
 self.play(Indicate(tex))
 self.wait()

 """
 def __init__(
 self,
 mobject: "Mobject",
 scale_factor: float = 1.2,
 color: str = YELLOW,
 rate_func: Callable[[float, Optional[float]], np.ndarray] = there_and_back,
 **kwargs
) -> None:
 self.color = color
 self.scale_factor = scale_factor
 super().__init__(mobject, rate_func=rate_func, **kwargs)

 def create_target(self) -> "Mobject":
 target = self.mobject.copy()
 target.scale(self.scale_factor)
 target.set_color(self.color)
 return target

class Flash(AnimationGroup):

```

```

"""Send out lines in all directions.

Parameters

point
 The center of the flash lines. If it is a :class:`.~Mobject` its center will be used.
line_length
 The length of the flash lines.
num_lines
 The number of flash lines.
flash_radius
 The distance from `point` at which the flash lines start.
line_stroke_width
 The stroke width of the flash lines.
color
 The color of the flash lines.
time_width
 The time width used for the flash lines. See :class:`.~ShowPassingFlash` for more details.
run_time
 The duration of the animation.
kwargs
 Additional arguments to be passed to the :class:`.~Succession` constructor

Examples

.. manim:: UsingFlash

 class UsingFlash(Scene):
 def construct(self):
 dot = Dot(color=YELLOW).shift(DOWN)
 self.add(Tex("Flash the dot below:"), dot)
 self.play(Flash(dot))
 self.wait()

.. manim:: FlashOnCircle

 class FlashOnCircle(Scene):
 def construct(self):
 radius = 2
 circle = Circle(radius)
 self.add(circle)
 self.play(Flash(
 circle, line_length=1,
 num_lines=30, color=RED,
 flash_radius=radius+SMALL_BUFF,
 time_width=0.3, run_time=2,
 rate_func = rush_from
))
 """

def __init__(
 self,
 point: Union[np.ndarray, Mobject],
 line_length: float = 0.2,
 num_lines: int = 12,
 flash_radius: float = 0.1,
 line_stroke_width: int = 3,
 color: str = YELLOW,
 time_width: float = 1,
 run_time: float = 1.0,
 **kwargs
) -> None:
 if isinstance(point, Mobject):
 self.point = point.get_center()
 else:
 self.point = point
 self.color = color
 self.line_length = line_length
 self.num_lines = num_lines
 self.flash_radius = flash_radius
 self.line_stroke_width = line_stroke_width
 self.run_time = run_time
 self.time_width = time_width
 self.animation_config = kwargs

 self.lines = self.create_lines()
 animations = self.create_line_anims()
 super().__init__(*animations, group=self.lines)

def create_lines(self) -> VGroup:
 lines = VGroup()
 for angle in np.arange(0, TAU, TAU / self.num_lines):
 line = Line(self.point, self.point + self.line_length * RIGHT)
 line.shift((self.flash_radius) * RIGHT)
 line.rotate(angle, about_point=self.point)
 lines.add(line)
 lines.set_color(self.color)

```

```

lines.set_stroke(width=self.line_stroke_width)
return lines

def create_line_anims(self) -> Iterable["ShowPassingFlash"]:
 return [
 ShowPassingFlash(
 line,
 time_width=self.time_width,
 run_time=self.run_time,
 **self.animation_config,
)
 for line in self.lines
]

class ShowPassingFlash(ShowPartial):
 """Show only a sliver of the VMobject each frame.

Parameters

mobject
 The mobject whose stroke is animated.
time_width
 The length of the sliver relative to the length of the stroke.

Examples

.. manim:: TimeWidthValues

 class TimeWidthValues(Scene):
 def construct(self):
 p = RegularPolygon(5, color=DARK_GRAY, stroke_width=6).scale(3)
 lbl = VMobject()
 self.add(p, lbl)
 p = p.copy().set_color(BLUE)
 for time_width in [0.2, 0.5, 1, 2]:
 lbl.become(Tex(r"\text{time}_width={\%.1f}" % time_width))
 self.play(ShowPassingFlash(
 p.copy().set_color(BLUE),
 run_time=2,
 time_width=time_width
))

```

See Also

```

:class:`~.Create`
```

"""

```

def __init__(self, mobject: "VMobject", time_width: float = 0.1, **kwargs) -> None:
 self.time_width = time_width
 super().__init__(mobject, remover=True, introducer=True, **kwargs)

def _get_bounds(self, alpha: float) -> Tuple[float]:
 tw = self.time_width
 upper = interpolate(0, 1 + tw, alpha)
 lower = upper - tw
 upper = min(upper, 1)
 lower = max(lower, 0)
 return (lower, upper)

def clean_up_from_scene(self, scene: "Scene") -> None:
 super().clean_up_from_scene(scene)
 for submob, start in self.get_all_families_zipped():
 submob.pointwiseBecomePartial(start, 0, 1)

class ShowPassingFlashWithThinningStrokeWidth(AnimationGroup):
 def __init__(self, vmobject, n_segments=10, time_width=0.1, remover=True, **kwargs):
 self.n_segments = n_segments
 self.time_width = time_width
 self.remover = remover
 max_stroke_width = vmobject.get_stroke_width()
 max_time_width = kwargs.pop("time_width", self.time_width)
 super().__init__(
 *(
 ShowPassingFlash(
 vmobject.copy().set_stroke(width=stroke_width),
 time_width=time_width,
 **kwargs,
)
 for stroke_width, time_width in zip(
 np.linspace(0, max_stroke_width, self.n_segments),
 np.linspace(max_time_width, 0, self.n_segments),
),
),
)

```

```

@deprecated(
 since="v0.15.0",
 until="v0.16.0",
 message="Use Create then FadeOut to achieve this effect.",
)
class ShowCreationThenFadeOut(Succession):
 def __init__(self, mobject: "Mobject", remover: bool = True, **kwargs) -> None:
 super().__init__(Create(mobject), FadeOut(mobject), remover=remover, **kwargs)

class ApplyWave(Homotopy):
 """Send a wave through the Mobject distorting it temporarily.

 Parameters

 mobject
 The mobject to be distorted.
 direction
 The direction in which the wave nudges points of the shape
 amplitude
 The distance points of the shape get shifted
 wave_func
 The function defining the shape of one wave flank.
 time_width
 The length of the wave relative to the width of the mobject.
 ripples
 The number of ripples of the wave
 run_time
 The duration of the animation.

 Examples

 .. manim:: ApplyingWaves

 class ApplyingWaves(Scene):
 def construct(self):
 tex = Tex("WaveWaveWaveWave").scale(2)
 self.play(ApplyWave(tex))
 self.play(ApplyWave(
 tex,
 direction=RIGHT,
 time_width=0.5,
 amplitude=0.3
))
 self.play(ApplyWave(
 tex,
 rate_func=linear,
 ripples=4
))

 """
 def __init__(
 self,
 mobject: "Mobject",
 direction: np.ndarray = UP,
 amplitude: float = 0.2,
 wave_func: Callable[[float], float] = smooth,
 time_width: float = 1,
 ripples: int = 1,
 run_time: float = 2,
 **kwargs
) -> None:
 x_min = mobject.get_left()[0]
 x_max = mobject.get_right()[0]
 vect = amplitude * normalize(direction)

 def wave(t):
 # Creates a wave with n ripples from a simple rate_func
 # This wave is build up as follows:
 # The time is split into 2*ripples phases. In every phase the amplitude
 # either rises to one or goes down to zero. Consecutive ripples will have
 # their amplitudes in oppising directions (first ripple from 0 to 1 to 0,
 # second from 0 to -1 to 0 and so on). This is how two ripples would be
 # divided into phases:

 # #####|#####
 # ## | ##
 # ## | ##
 # ##### | ##### | #####
 # | | ## |
 # | | ## |
 # | | #####

```

```

However, this looks weird in the middle between two ripples. Therefore the
middle phases do actually use only one appropriately scaled version of the
rate like this:

1 / 4 Time | 2 / 4 Time | 1 / 4 Time
#####|#####|#####
| ## |
| ## |
#####| # | #####|
| ## |
| ### |
| #####|#####|#####
| ######|######

Mirrored looks better in the way the wave is used.
t = 1 - t

Clamp input
if t >= 1 or t <= 0:
 return 0

phases = ripples * 2
phase = int(t * phases)
if phase == 0:
 # First rising ripple
 return wave_func(t * phases)
elif phase == phases - 1:
 # last ripple. Rising or falling depending on the number of ripples
 # The (ripples % 2)-term is used to make this distinction.
 t -= phase / phases # Time relative to the phase
 return (1 - wave_func(t * phases)) * (2 * (ripples % 2) - 1)
else:
 # Longer phases:
 phase = int((phase - 1) / 2)
 t -= (2 * phase + 1) / phases

 # Similar to last ripple:
 return (1 - 2 * wave_func(t * ripples)) * (1 - 2 * ((phase) % 2))

def homotopy(
 x: float,
 y: float,
 z: float,
 t: float,
) -> Tuple[float, float, float]:
 upper = interpolate(0, 1 + time_width, t)
 lower = upper - time_width
 relative_x = inverse_interpolate(x_min, x_max, x)
 wave_phase = inverse_interpolate(lower, upper, relative_x)
 nudge = wave(wave_phase) * vect
 return np.array([x, y, z]) + nudge

super().__init__(homotopy, mobject, run_time=run_time, **kwargs)

class Wiggle(Animation):
 """Wiggle a Mobject.

 Parameters

 mobject
 The mobject to wiggle.
 scale_value
 The factor by which the mobject will be temporarily scaled.
 rotation_angle
 The wiggle angle.
 n_wiggles
 The number of wiggles.
 scale_about_point
 The point about which the mobject gets scaled.
 rotate_about_point
 The point around which the mobject gets rotated.
 run_time
 The duration of the animation

 Examples

 .. manim:: ApplyingWaves

 class ApplyingWaves(Scene):
 def construct(self):
 tex = Tex("Wiggle").scale(3)
 self.play(Wiggle(tex))
 self.wait()

 """

```

```

def __init__(

 self,

 mobobject: "Mobject",

 scale_value: float = 1.1,

 rotation_angle: float = 0.01 * TAU,

 n_wiggles: int = 6,

 scale_about_point: Optional[np.ndarray] = None,

 rotate_about_point: Optional[np.ndarray] = None,

 run_time: float = 2,

 **kwargs
) -> None:

 self.scale_value = scale_value

 self.rotation_angle = rotation_angle

 self.n_wiggles = n_wiggles

 self.scale_about_point = scale_about_point

 self.rotate_about_point = rotate_about_point

 super().__init__(mobobject, run_time=run_time, **kwargs)

def get_scale_about_point(self) -> np.ndarray:

 if self.scale_about_point is None:

 return self.mobobject.get_center()

 return self.scale_about_point

def get_rotate_about_point(self) -> np.ndarray:

 if self.rotate_about_point is None:

 return self.mobobject.get_center()

 return self.rotate_about_point

def interpolate_submobject(

 self,

 submobject: "Mobject",

 starting_submobject: "Mobject",

 alpha: float,
) -> None:

 submobject.points[:, :] = starting_submobject.points

 submobject.scale(

 interpolate(1, self.scale_value, there_and_back(alpha)),

 about_point=self.get_scale_about_point(),

)

 submobject.rotate(

 wiggle(alpha, self.n_wiggles) * self.rotation_angle,

 about_point=self.get_rotate_about_point(),

)

class Circumscribe(Succession):

 """Draw a temporary line surrounding the mobobject.

 Parameters

 mobobject

 The mobobject to be circumscribed.

 shape

 The shape with which to surround the given mobobject. Should be either

 :class:`~.Rectangle` or :class:`~.Circle`

 fade_in

 Whether to make the surrounding shape to fade in. It will be drawn otherwise.

 fade_out

 Whether to make the surrounding shape to fade out. It will be undrawn otherwise.

 time_width

 The time_width of the drawing and undrawing. Gets ignored if either `fade_in` or `fade_out` is `True`.

 buff

 The distance between the surrounding shape and the given mobobject.

 color

 The color of the surrounding shape.

 run_time

 The duration of the entire animation.

 kwargs

 Additional arguments to be passed to the :class:`~.Succession` constructor

 Examples

.. manim:: UsingCircumscribe

 class UsingCircumscribe(Scene):

 def construct(self):

 lbl = Tex(r"Circum-\\"\\scribe").scale(2)

 self.add(lbl)

 self.play(Circumscribe(lbl))

 self.play(Circumscribe(lbl, Circle))

 self.play(Circumscribe(lbl, fade_out=True))

 self.play(Circumscribe(lbl, time_width=2))

 self.play(Circumscribe(lbl, Circle, True))

"""

```

```

def __init__(

 self,

 mobject: Mobject,

 shape: Type = Rectangle,

 fade_in=False,

 fade_out=False,

 time_width=0.3,

 buff: float = SMALL_BUFF,

 color: Color = YELLOW,

 run_time=1,

 stroke_width=DEFAULT_STROKE_WIDTH,

 **kwargs

):

 if shape is Rectangle:

 frame = SurroundingRectangle(

 mobject,

 color,

 buff,

 stroke_width=stroke_width,

)

 elif shape is Circle:

 frame = Circle(color=color, stroke_width=stroke_width).surround(

 mobject,

 buffer_factor=1,

)

 radius = frame.width / 2

 frame.scale((radius + buff) / radius)

 else:

 raise ValueError("shape should be either Rectangle or Circle.")

 if fade_in and fade_out:

 super().__init__(

 FadeIn(frame, run_time=run_time / 2),

 FadeOut(frame, run_time=run_time / 2),

 **kwargs,

)

 elif fade_in:

 frame.reverse_direction()

 super().__init__(

 FadeIn(frame, run_time=run_time / 2),

 Uncreate(frame, run_time=run_time / 2),

 **kwargs,

)

 elif fade_out:

 super().__init__(

 Create(frame, run_time=run_time / 2),

 FadeOut(frame, run_time=run_time / 2),

 **kwargs,

)

 else:

 super().__init__(

 ShowPassingFlash(frame, time_width, run_time=run_time), **kwargs

)

```

# value\_tracker.py

```
"""Simple mobjects that can be used for storing (and updating) a value."""

from __future__ import annotations

__all__ = ["ValueTracker", "ComplexValueTracker"]

import numpy as np

from manim.mobject.mobject import Mobject
from manim.mobject.opengl.opengl_compatibility import ConvertToOpenGL
from manim.utils.paths import straight_path

class ValueTracker(Mobject, metaclass=ConvertToOpenGL):
 """A mobject that can be used for tracking (real-valued) parameters.
 Useful for animating parameter changes.

 Not meant to be displayed. Instead the position encodes some
 number, often one which another animation or continual_animation
 uses for its update function, and by treating it as a mobject it can
 still be animated and manipulated just like anything else.

 This value changes continuously when animated using the :attr:`animate` syntax.

Examples

.. manim:: ValueTrackerExample

 class ValueTrackerExample(Scene):
 def construct(self):
 number_line = NumberLine()
 pointer = Vector(DOWN)
 label = MathTex("x").add_updater(lambda m: m.next_to(pointer, UP))

 tracker = ValueTracker(0)
 pointer.add_updater(
 lambda m: m.next_to(
 number_line.n2p(tracker.get_value()),
 UP
)
)
 self.add(number_line, pointer, label)
 tracker += 1.5
 self.wait(1)
 tracker -= 4
 self.wait(0.5)
 self.play(tracker.animate.set_value(5)),
 self.wait(0.5)
 self.play(tracker.animate.set_value(3))
 self.play(tracker.animate.increment_value(-2))
 self.wait(0.5)

.. note::

 You can also link ValueTrackers to updaters. In this case, you have to make sure that the
 ValueTracker is added to the scene by ``add``

.. manim:: ValueTrackerExample

 class ValueTrackerExample(Scene):
 def construct(self):
 tracker = ValueTracker(0)
 label = Dot(radius=3).add_updater(lambda x : x.set_x(tracker.get_value()))
 self.add(label)
 self.add(tracker)
 tracker.add_updater(lambda mobject, dt: mobject.increment_value(dt))
 self.wait(2)

"""

def __init__(self, value=0, **kwargs):
 super().__init__(**kwargs)
 self.set_points(np.zeros((1, 3)))
 self.set_value(value)

def get_value(self) -> float:
 """Get the current value of this ValueTracker."""
 return self.points[0, 0]

def set_value(self, value: float):
```

```

 """Sets a new scalar value to the ValueTracker"""
 self.points[0, 0] = value
 return self

def increment_value(self, d_value: float):
 """Increments (adds) a scalar value to the ValueTracker"""
 self.set_value(self.get_value() + d_value)
 return self

def __bool__(self):
 """Return whether the value of this value tracker evaluates as true."""
 return bool(self.get_value())

def __iadd__(self, d_value: float):
 """`+=` syntax to increment the value of the ValueTracker"""
 self.increment_value(d_value)
 return self

def __ifloordiv__(self, d_value: float):
 """Set the value of this value tracker to the floor division of the current value by ``d_value``."""
 self.set_value(self.get_value() // d_value)
 return self

def __imod__(self, d_value: float):
 """Set the value of this value tracker to the current value modulo ``d_value``."""
 self.set_value(self.get_value() % d_value)
 return self

def __imul__(self, d_value: float):
 """Set the value of this value tracker to the product of the current value and ``d_value``."""
 self.set_value(self.get_value() * d_value)
 return self

def __ipow__(self, d_value: float):
 """Set the value of this value tracker to the current value raised to the power of ``d_value``."""
 self.set_value(self.get_value() ** d_value)
 return self

def __isub__(self, d_value: float):
 """`-=` syntax to decrement the value of the ValueTracker"""
 self.increment_value(-d_value)
 return self

def __itruediv__(self, d_value: float):
 """Sets the value of this value tracker to the current value divided by ``d_value``."""
 self.set_value(self.get_value() / d_value)
 return self

def interpolate(self, mobject1, mobject2, alpha, path_func=straight_path()):
 """
 Turns self into an interpolation between mobject1 and mobject2.

 self.set_points(path_func(mobject1.points, mobject2.points, alpha))
 return self
 """

class ComplexValueTracker(ValueTracker):
 """Tracks a complex-valued parameter.

When the value is set through :attr:`animate`, the value will take a straight path from the source point to the destination point.

Examples

.. manim:: ComplexValueTrackerExample

 class ComplexValueTrackerExample(Scene):
 def construct(self):
 tracker = ComplexValueTracker(-2+1j)
 dot = Dot().add_updater(
 lambda x: x.move_to(tracker.points)
)

 self.add(NumberPlane(), dot)

 self.play(tracker.animate.set_value(3+2j))
 self.play(tracker.animate.set_value(tracker.get_value() * 1j))
 self.play(tracker.animate.set_value(tracker.get_value() - 2j))
 self.play(tracker.animate.set_value(tracker.get_value() / (-2 + 3j)))
 """

def get_value(self):
 """Get the current value of this value tracker as a complex number.

The value is internally stored as a points array [a, b, 0]. This can be accessed directly to represent the value geometrically, see the usage example."""

```

```
 return complex(*self.points[0, :2])

def set_value(self, z):
 """Sets a new complex value to the ComplexValueTracker"""
 z = complex(z)
 self.points[0, :2] = (z.real, z.imag)
 return self
```

# vectorized\_mobject\_rendering.py

```
from __future__ import annotations

import collections

import numpy as np

from ..utils import opengl
from ..utils.space_ops import cross2d, earclip_triangulation
from .shader import Shader

def build_matrix_lists(mob):
 root_hierarchical_matrix = mob.hierarchical_model_matrix()
 matrix_to_mobject_list = collections.defaultdict(list)
 if mob.has_points():
 matrix_to_mobject_list[tuple(root_hierarchical_matrix.ravel())].append(mob)
 mobject_to_hierarchical_matrix = {mob: root_hierarchical_matrix}
 dfs = [mob]
 while dfs:
 parent = dfs.pop()
 for child in parent.subobjects:
 child_hierarchical_matrix = (
 mobject_to_hierarchical_matrix[parent] @ child.model_matrix
)
 mobject_to_hierarchical_matrix[child] = child_hierarchical_matrix
 if child.has_points():
 matrix_to_mobject_list[tuple(child_hierarchical_matrix.ravel())].append(
 child,
)
 dfs.append(child)
 return matrix_to_mobject_list

def render_opengl_vectorized_mobject_fill(renderer, mobject):
 matrix_to_mobject_list = build_matrix_lists(mobject)

 for matrix_tuple, mobject_list in matrix_to_mobject_list.items():
 model_matrix = np.array(matrix_tuple).reshape((4, 4))
 render_mobject_fills_with_matrix(renderer, model_matrix, mobject_list)

def render_mobject_fills_with_matrix(renderer, model_matrix, mobjects):
 # Precompute the total number of vertices for which to reserve space.
 # Note that triangulate_mobject() will cache its results.
 total_size = 0
 for submob in mobjects:
 total_size += triangulate_mobject(submob).shape[0]

 attributes = np.empty(
 total_size,
 dtype=[
 ("in_vert", np.float32, (3,)),
 ("in_color", np.float32, (4,)),
 ("texture_coords", np.float32, (2,)),
 ("texture_mode", np.int32),
],
)

 write_offset = 0
 for submob in mobjects:
 if not submob.has_points():
 continue
 mobject_triangulation = triangulate_mobject(submob)
 end_offset = write_offset + mobject_triangulation.shape[0]
 attributes[write_offset:end_offset] = mobject_triangulation
 attributes["in_color"][write_offset:end_offset] = np.repeat(
 submob.fill_rgba,
 mobject_triangulation.shape[0],
 axis=0,
)
 write_offset = end_offset

 fill_shader = Shader(renderer.context, name="vectorized_mobject_fill")
 fill_shader.set_uniform(
 "u_model_view_matrix",
 opengl.matrix_to_shader_input(
 renderer.camera.unformatted_view_matrix @ model_matrix,
),
)
 fill_shader.set_uniform(
 "u_projection_matrix",
```

```

 renderer.scene.camera.projection_matrix,
)

vbo = renderer.context.buffer(attributes.tobytes())
vao = renderer.context.simple_vertex_array(
 fill_shader.shader_program,
 vbo,
 *attributes.dtype.names,
)
vao.render()
vao.release()
vbo.release()

def triangulate_mobject(mob):
 if not mob.needs_new_triangulation:
 return mob.triangulation

 # Figure out how to triangulate the interior to know
 # how to send the points as to the vertex shader.
 # First triangles come directly from the points
 # normal_vector = mob.get_unit_normal()
 points = mob.points

 b0s = points[0::3]
 b1s = points[1::3]
 b2s = points[2::3]
 v01s = b1s - b0s
 v12s = b2s - b1s

 crosses = cross2d(v01s, v12s)
 convexities = np.sign(crosses)
 if mob.orientation == 1:
 concave_parts = convexities > 0
 convex_parts = convexities <= 0
 else:
 concave_parts = convexities < 0
 convex_parts = convexities >= 0

 # These are the vertices to which we'll apply a polygon triangulation
 atol = mob.tolerance_for_point_equality
 end_of_loop = np.zeros(len(b0s), dtype=bool)
 end_of_loop[:-1] = (np.abs(b2s[:-1] - b0s[1:]) > atol).any(1)
 end_of_loop[-1] = True

 indices = np.arange(len(points), dtype=int)
 inner_vert_indices = np.hstack(
 [
 indices[0::3],
 indices[1::3][concave_parts],
 indices[2::3][end_of_loop],
],
)
 inner_vert_indices.sort()
 rings = np.arange(1, len(inner_vert_indices) + 1)[inner_vert_indices % 3 == 2]

 # Triangulate
 inner_verts = points[inner_vert_indices]
 inner_tri_indices = inner_vert_indices[earclip_triangulation(inner_verts, rings)]

 bezier_triangle_indices = np.reshape(indices, (-1, 3))
 concave_triangle_indices = np.reshape(bezier_triangle_indices[concave_parts], (-1))
 convex_triangle_indices = np.reshape(bezier_triangle_indices[convex_parts], (-1))

 points = points[
 np.hstack(
 [
 concave_triangle_indices,
 convex_triangle_indices,
 inner_tri_indices,
],
),
]
 texture_coords = np.tile(
 [
 [0.0, 0.0],
 [0.5, 0.0],
 [1.0, 1.0],
],
 (points.shape[0] // 3, 1),
)
 texture_mode = np.hstack(
 (
 np.ones(concave_triangle_indices.shape[0]),
 -1 * np.ones(convex_triangle_indices.shape[0]),
 np.zeros(inner_tri_indices.shape[0]),
),
),

```

```

)
attributes = np.zeros(
 points.shape[0],
 dtype=[

 ("in_vert", np.float32, (3,)),
 ("in_color", np.float32, (4,)),
 ("texture_coords", np.float32, (2,)),
 ("texture_mode", np.int32),
],
)
attributes["in_vert"] = points
attributes["texture_coords"] = texture_coords
attributes["texture_mode"] = texture_mode

mob.triangulation = attributes
mob.needs_new_triangulation = False

return attributes

def render_opengl_vectorized_mobject_stroke(renderer, mobject):
 matrix_to_mobject_list = build_matrix_lists(mobject)
 for matrix_tuple, mobject_list in matrix_to_mobject_list.items():
 model_matrix = np.array(matrix_tuple).reshape((4, 4))
 render_mobject_strokes_with_matrix(renderer, model_matrix, mobject_list)

def render_mobject_strokes_with_matrix(renderer, model_matrix, mobjects):
 # Precompute the total number of vertices for which to reserve space.
 total_size = 0
 for submob in mobjects:
 total_size += submob.points.shape[0]

 points = np.empty((total_size, 3))
 colors = np.empty((total_size, 4))
 widths = np.empty(total_size)

 write_offset = 0
 for submob in mobjects:
 if not submob.has_points():
 continue
 end_offset = write_offset + submob.points.shape[0]

 points[write_offset:end_offset] = submob.points
 if submob.stroke_rgba.shape[0] == points[write_offset:end_offset].shape[0]:
 colors[write_offset:end_offset] = submob.stroke_rgba
 else:
 colors[write_offset:end_offset] = np.repeat(
 submob.stroke_rgba,
 submob.points.shape[0],
 axis=0,
)
 widths[write_offset:end_offset] = np.repeat(
 submob.stroke_width,
 submob.points.shape[0],
)
 write_offset = end_offset

 stroke_data = np.zeros(
 len(points),
 dtype=[

 # ("previous_curve", np.float32, (3, 3)),
 ("current_curve", np.float32, (3, 3)),
 # ("next_curve", np.float32, (3, 3)),
 ("tile_coordinate", np.float32, (2,)),
 ("in_color", np.float32, (4,)),
 ("in_width", np.float32),
],
)

 stroke_data["in_color"] = colors
 stroke_data["in_width"] = widths
 curves = np.reshape(points, (-1, 3, 3))
 # stroke_data["previous_curve"] = np.repeat(np.roll(curves, 1, axis=0), 3, axis=0)
 stroke_data["current_curve"] = np.repeat(curves, 3, axis=0)
 # stroke_data["next_curve"] = np.repeat(np.roll(curves, -1, axis=0), 3, axis=0)

 # Repeat each vertex in order to make a tile.
 stroke_data = np.tile(stroke_data, 2)
 stroke_data["tile_coordinate"] = np.vstack(
 (
 np.tile(
 [
 [0.0, 0.0],
 [0.0, 1.0],
 [1.0, 1.0],
],
 3
)
)
)

```

```

],
 (len(points) // 3, 1),
),
 np.tile(
 [
 [0.0, 0.0],
 [1.0, 0.0],
 [1.0, 1.0],
],
 (len(points) // 3, 1),
),
),
)
)

shader = Shader(renderer.context, "vectorized_mobject_stroke")
shader.set_uniform(
 "u_model_view_matrix",
 opengl.matrix_to_shader_input(
 renderer.camera.unformatted_view_matrix @ model_matrix,
),
)
shader.set_uniform("u_projection_matrix", renderer.scene.camera.projection_matrix)
shader.set_uniform("manim_unit_normal", tuple(-mobjects[0].unit_normal[0]))

vbo = renderer.context.buffer(stroke_data.tobytes())
vao = renderer.context.simple_vertex_array(
 shader.shader_program, vbo, *stroke_data.dtype.names
)
renderer.frame_buffer_object.use()
vao.render()
vao.release()
vbo.release()

```

# vector\_space\_scene.py

```
"""A scene suitable for vector spaces."""

from __future__ import annotations

__all__ = ["VectorScene", "LinearTransformationScene"]

from typing import Callable

import numpy as np
from colour import Color

from manim.mobject.geometry.arc import Dot
from manim.mobject.geometry.line import Arrow, Line, Vector
from manim.mobject.geometry.polygram import Rectangle
from manim.mobject.graphing.coordinate_systems import Axes, NumberPlane
from manim.mobject.opengl.opengl_mobject import OpenGLObject
from manim.mobject.text.tex_mobject import MathTex, Tex
from manim.utils.config_ops import update_dict_recursively

from .. import config
from ..animation.animation import Animation
from ..animation.creation import Create, Write
from ..animation.fading import FadeOut
from ..animation.growing import GrowArrow
from ..animation.transform import ApplyFunction, ApplyPointwiseFunction, Transform
from ..constants import *
from ..mobject.matrix import Matrix
from ..mobject.mobject import Mobject
from ..mobject.types.vectorized_mobject import VGroup, VMobject
from ..scene.scene import Scene
from ..utils.color import BLUE_D, GREEN_C, GREY, RED_C, WHITE, YELLOW
from ..utils.rate_functions import rush_from, rush_into
from ..utils.space_ops import angle_of_vector

X_COLOR = GREEN_C
Y_COLOR = RED_C
Z_COLOR = BLUE_D

TODO: Much of this scene type seems dependent on the coordinate system chosen.
That is, being centered at the origin with grid units corresponding to the
arbitrary space units. Change it!
#
Also, methods I would have thought of as getters, like coords_to_vector, are
actually doing a lot of animating.

class VectorScene(Scene):
 def __init__(self, basis_vector_stroke_width=6, **kwargs):
 super().__init__(**kwargs)
 self.basis_vector_stroke_width = basis_vector_stroke_width

 def add_plane(self, animate: bool = False, **kwargs):
 """
 Adds a NumberPlane object to the background.

 Parameters

 animate
 Whether or not to animate the addition of the plane via Create.
 **kwargs
 Any valid keyword arguments accepted by NumberPlane.

 Returns

 NumberPlane
 The NumberPlane object.
 """
 plane = NumberPlane(**kwargs)
 if animate:
 self.play(Create(plane, lag_ratio=0.5))
 self.add(plane)
 return plane

 def add_axes(self, animate: bool = False, color: bool = WHITE, **kwargs):
 """
 Adds a pair of Axes to the Scene.

 Parameters

 animate
 Whether or not to animate the addition of the axes through Create.
 color

```

```

 The color of the axes. Defaults to WHITE.
"""
axes = Axes(color=color, axis_config={"unit_size": 1})
if animate:
 self.play(Create(axes))
self.add(axes)
return axes

def lock_in_faded_grid(self, dimness: float = 0.7, axes_dimness: float = 0.5):
"""
This method freezes the NumberPlane and Axes that were already
in the background, and adds new, manipulatable ones to the foreground.

Parameters

dimness
 The required dimness of the NumberPlane

axes_dimness
 The required dimness of the Axes.
"""
plane = self.add_plane()
axes = plane.get_axes()
plane.fade(dimness)
axes.set_color(WHITE)
axes.fade(axes_dimness)
self.add(axes)

self.renderer.update_frame()
self.renderer.camera = Camera(self.renderer.get_frame())
self.clear()

def get_vector(self, numerical_vector: np.ndarray | list | tuple, **kwargs):
"""
Returns an arrow on the Plane given an input numerical vector.

Parameters

numerical_vector
 The Vector to plot.
**kwargs
 Any valid keyword argument of Arrow.

Returns

Arrow
 The Arrow representing the Vector.
"""
return Arrow(
 self.plane.coords_to_point(0, 0),
 self.plane.coords_to_point(*numerical_vector[:2]),
 buff=0,
 **kwargs,
)

def add_vector(
 self,
 vector: Arrow | list | tuple | np.ndarray,
 color: str = YELLOW,
 animate: bool = True,
 **kwargs,
):
"""
Returns the Vector after adding it to the Plane.

Parameters

vector
 It can be a pre-made graphical vector, or the
 coordinates of one.

color
 The string of the hex color of the vector.
 This is only taken into consideration if
 'vector' is not an Arrow. Defaults to YELLOW.

animate
 Whether or not to animate the addition of the vector
 by using GrowArrow

**kwargs
 Any valid keyword argument of Arrow.
 These are only considered if vector is not
 an Arrow.

Returns

```

```

Arrow
 The arrow representing the vector.
"""
if not isinstance(vector, Arrow):
 vector = Vector(vector, color=color, **kwargs)
if animate:
 self.play(GrowArrow(vector))
self.add(vector)
return vector

def write_vector_coordinates(self, vector: Arrow, **kwargs):
"""
Returns a column matrix indicating the vector coordinates,
after writing them to the screen.

Parameters

vector
 The arrow representing the vector.

**kwargs
 Any valid keyword arguments of :meth:`~.Vector.coordinate_label`:

Returns

:class:`.Matrix`
 The column matrix representing the vector.
"""

coords = vector.coordinate_label(**kwargs)
self.play(Write(coords))
return coords

def get_basis_vectors(self, i_hat_color: str = X_COLOR, j_hat_color: str = Y_COLOR):
"""
Returns a VGroup of the Basis Vectors (1,0) and (0,1)

Parameters

i_hat_color
 The hex colour to use for the basis vector in the x direction

j_hat_color
 The hex colour to use for the basis vector in the y direction

Returns

VGroup
 VGroup of the Vector Mobjects representing the basis vectors.
"""

return VGroup(
 *(
 Vector(vect, color=color, stroke_width=self.basis_vector_stroke_width)
 for vect, color in [([1, 0], i_hat_color), ([0, 1], j_hat_color)]
)
)

def get_basis_vector_labels(self, **kwargs):
"""
Returns naming labels for the basis vectors.

Parameters

**kwargs
 Any valid keyword arguments of get_vector_label:
 vector,
 label (str,MathTex),
 at_tip (bool=False),
 direction (str="left"),
 rotate (bool),
 color (str),
 label_scale_factor=VECTOR_LABEL_SCALE_FACTOR (int, float),
"""

i_hat, j_hat = self.get_basis_vectors()
return VGroup(
 *(
 self.get_vector_label(
 vect, label, color=color, label_scale_factor=1, **kwargs
)
 for vect, label, color in [
 (i_hat, "\hat{\imath}", X_COLOR),
 (j_hat, "\hat{\jmath}", Y_COLOR),
]
)
)

def get_vector_label(
 self,

```

```

vector: Vector,
label,
at_tip: bool = False,
direction: str = "left",
rotate: bool = False,
color: str | None = None,
label_scale_factor: float = LARGE_BUFF - 0.2,
):
"""
 Returns naming labels for the passed vector.

Parameters

vector
 Vector Object for which to get the label.

at_tip
 Whether or not to place the label at the tip of the vector.

direction
 If the label should be on the "left" or right of the vector.
rotate
 Whether or not to rotate it to align it with the vector.
color
 The color to give the label.
label_scale_factor
 How much to scale the label by.

Returns

MathTex
 The MathTex of the label.
"""

if not isinstance(label, MathTex):
 if len(label) == 1:
 label = "\\\vec{\\textbf{\{}}%s\}}" % label
 label = MathTex(label)
 if color is None:
 color = vector.get_color()
 label.set_color(color)
label.scale(label_scale_factor)
label.add_background_rectangle()

if at_tip:
 vect = vector.get_vector()
 vect /= np.linalg.norm(vect)
 label.next_to(vector.get_end(), vect, buff=SMALL_BUFF)
else:
 angle = vector.get_angle()
 if not rotate:
 label.rotate(-angle, about_point=ORIGIN)
 if direction == "left":
 label.shift(-label.get_bottom() + 0.1 * UP)
 else:
 label.shift(-label.get_top() + 0.1 * DOWN)
 label.rotate(angle, about_point=ORIGIN)
 label.shift((vector.get_end() - vector.get_start()) / 2)
return label

def label_vector(
 self, vector: Vector, label: MathTex | str, animate: bool = True, **kwargs
):
"""
 Shortcut method for creating, and animating the addition of
 a label for the vector.

Parameters

vector
 The vector for which the label must be added.

label
 The MathTex/string of the label.

animate
 Whether or not to animate the labelling w/ Write

**kwargs
 Any valid keyword argument of get_vector_label

Returns

:class:`~.MathTex`
 The MathTex of the label.
"""

label = self.get_vector_label(vector, label, **kwargs)
if animate:

```

```

 self.play(Write(label, run_time=1))
 self.add(label)
 return label

def position_x_coordinate(
 self,
 x_coord,
 x_line,
 vector,
):
 # TODO Write DocStrings for this.
 x_coord.next_to(x_line, -np.sign(vector[1]) * UP)
 x_coord.set_color(X_COLOR)
 return x_coord

def position_y_coordinate(
 self,
 y_coord,
 y_line,
 vector,
):
 # TODO Write DocStrings for this.
 y_coord.next_to(y_line, np.sign(vector[0]) * RIGHT)
 y_coord.set_color(Y_COLOR)
 return y_coord

def coords_to_vector(
 self,
 vector: np.ndarray | list | tuple,
 coords_start: np.ndarray | list | tuple = 2 * RIGHT + 2 * UP,
 clean_up: bool = True,
):
 """
 This method writes the vector as a column matrix (henceforth called the label),
 takes the values in it one by one, and form the corresponding
 lines that make up the x and y components of the vector. Then, an
 Vector() based vector is created between the lines on the Screen.

 Parameters

 vector
 The vector to show.

 coords_start
 The starting point of the location of
 the label of the vector that shows it
 numerically.
 Defaults to 2 * RIGHT + 2 * UP or (2,2)

 clean_up
 Whether or not to remove whatever
 this method did after it's done.

 """
 starting_mobjects = list(self.mobjects)
 array = Matrix(vector)
 array.shift(coords_start)
 arrow = Vector(vector)
 x_line = Line(ORIGIN, vector[0] * RIGHT)
 y_line = Line(x_line.get_end(), arrow.get_end())
 x_line.set_color(X_COLOR)
 y_line.set_color(Y_COLOR)
 x_coord, y_coord = array.get_mob_matrix().flatten()

 self.play(Write(array, run_time=1))
 self.wait()
 self.play(
 ApplyFunction(
 lambda x: self.position_x_coordinate(x, x_line, vector),
 x_coord,
),
)
 self.play(Create(x_line))
 animations = [
 ApplyFunction(
 lambda y: self.position_y_coordinate(y, y_line, vector),
 y_coord,
),
 FadeOut(array.get_brackets()),
]
 self.play(*animations)
 y_coord, _ = (anim.mob for anim in animations)
 self.play(Create(y_line))
 self.play(Create(arrow))
 self.wait()
 if clean_up:
 self.clear()
 self.add(*starting_mobjects)

```

```

def vector_to_coords(
 self,
 vector: np.ndarray | list | tuple,
 integer_labels: bool = True,
 clean_up: bool = True,
):
 """
 This method displays vector as a Vector() based vector, and then shows
 the corresponding lines that make up the x and y components of the vector.
 Then, a column matrix (henceforth called the label) is created near the
 head of the Vector.

 Parameters

 vector
 The vector to show.

 integer_labels
 Whether or not to round the value displayed.
 in the vector's label to the nearest integer

 clean_up
 Whether or not to remove whatever
 this method did after it's done.

 """
 starting_mobjects = list(self.mobjects)
 show_creation = False
 if isinstance(vector, Arrow):
 arrow = vector
 vector = arrow.get_end()[:2]
 else:
 arrow = Vector(vector)
 show_creation = True
 array = arrow.coordinate_label(integer_labels=integer_labels)
 x_line = Line(ORIGIN, vector[0] * RIGHT)
 y_line = Line(x_line.get_end(), arrow.get_end())
 x_line.set_color(X_COLOR)
 y_line.set_color(Y_COLOR)
 x_coord, y_coord = array.get_entries()
 x_coord_start = self.position_x_coordinate(x_coord.copy(), x_line, vector)
 y_coord_start = self.position_y_coordinate(y_coord.copy(), y_line, vector)
 brackets = array.get_brackets()

 if show_creation:
 self.play(Create(arrow))
 self.play(Create(x_line), Write(x_coord_start), run_time=1)
 self.play(Create(y_line), Write(y_coord_start), run_time=1)
 self.wait()
 self.play(
 Transform(x_coord_start, x_coord, lag_ratio=0),
 Transform(y_coord_start, y_coord, lag_ratio=0),
 Write(brackets, run_time=1),
)
 self.wait()

 self.remove(x_coord_start, y_coord_start, brackets)
 self.add(array)
 if clean_up:
 self.clear()
 self.add(*starting_mobjects)
 return array, x_line, y_line

def show_ghost_movement(self, vector: Arrow | list | tuple | np.ndarray):
 """
 This method plays an animation that partially shows the entire plane moving
 in the direction of a particular vector. This is useful when you wish to
 convey the idea of mentally moving the entire plane in a direction, without
 actually moving the plane.

 Parameters

 vector
 The vector which indicates the direction of movement.

 """
 if isinstance(vector, Arrow):
 vector = vector.get_end() - vector.get_start()
 elif len(vector) == 2:
 vector = np.append(np.array(vector), 0.0)
 x_max = int(config["frame_x_radius"] + abs(vector[0]))
 y_max = int(config["frame_y_radius"] + abs(vector[1]))
 dots = VMobject(
 *(
 Dot(x * RIGHT + y * UP)
 for x in range(-x_max, x_max)
 for y in range(-y_max, y_max)
)
)

```

```

)
dots.set_fill(BLACK, opacity=0)
dots_halfway = dots.copy().shift(vector / 2).set_fill(WHITE, 1)
dots_end = dots.copy().shift(vector)

self.play(Transform(dots, dots_halfway, rate_func=rush_into))
self.play(Transform(dots, dots_end, rate_func=rush_from))
self.remove(dots)

class LinearTransformationScene(VectorScene):
 """
 This scene contains special methods that make it
 especially suitable for showing linear transformations.

 Parameters

 include_background_plane
 Whether or not to include the background plane in the scene.
 include_foreground_plane
 Whether or not to include the foreground plane in the scene.
 background_plane_kwargs
 Parameters to be passed to :class:`NumberPlane` to adjust the background plane.
 foreground_plane_kwargs
 Parameters to be passed to :class:`NumberPlane` to adjust the foreground plane.
 show_coordinates
 Whether or not to include the coordinates for the background plane.
 show_basis_vectors
 Whether to show the basis x -axis $\rightarrow \hat{i}$ and y -axis $\rightarrow \hat{j}$ vectors.
 basis_vector_stroke_width
 The ``stroke_width`` of the basis vectors.
 i_hat_color
 The color of the `` \hat{i} `` vector.
 j_hat_color
 The color of the `` \hat{j} `` vector.
 leave_ghost_vectors
 Indicates the previous position of the basis vectors following a transformation.

 Examples

 .. manim:: LinearTransformationSceneExample

 class LinearTransformationSceneExample(LinearTransformationScene):
 def __init__(self):
 LinearTransformationScene.__init__(
 self,
 show_coordinates=True,
 leave_ghost_vectors=True,
)

 def construct(self):
 matrix = [[1, 1], [0, 1]]
 self.apply_matrix(matrix)
 self.wait()
 """
 def __init__(self,
 include_background_plane: bool = True,
 include_foreground_plane: bool = True,
 background_plane_kwargs: dict | None = None,
 foreground_plane_kwargs: dict | None = None,
 show_coordinates: bool = False,
 show_basis_vectors: bool = True,
 basis_vector_stroke_width: float = 6,
 i_hat_color: Color = X_COLOR,
 j_hat_color: Color = Y_COLOR,
 leave_ghost_vectors: bool = False,
 **kwargs,
):
 super().__init__(**kwargs)

 self.include_background_plane = include_background_plane
 self.include_foreground_plane = include_foreground_plane
 self.show_coordinates = show_coordinates
 self.show_basis_vectors = show_basis_vectors
 self.basis_vector_stroke_width = basis_vector_stroke_width
 self.i_hat_color = i_hat_color
 self.j_hat_color = j_hat_color
 self.leave_ghost_vectors = leave_ghost_vectors
 self.background_plane_kwargs = {
 "color": GREY,
 "axis_config": {
 "color": GREY,
 },
 "background_line_style": {

```

```

 "stroke_color": GREY,
 "stroke_width": 1,
 },
}

self.foreground_plane_kw_args = {
 "x_range": np.array([-config["frame_width"], config["frame_width"], 1.0]),
 "y_range": np.array([-config["frame_width"], config["frame_width"], 1.0]),
 "faded_line_ratio": 1,
}
)

self.update_default_configs(
 (self.foreground_plane_kw_args, self.background_plane_kw_args),
 (foreground_plane_kw_args, background_plane_kw_args),
)
)

@staticmethod
def update_default_configs(default_configs, passed_configs):
 for default_config, passed_config in zip(default_configs, passed_configs):
 if passed_config is not None:
 update_dict_recursively(default_config, passed_config)

def setup(self):
 # The has_already_setup attr is to not break all the old Scenes
 if hasattr(self, "has_already_setup"):
 return
 self.has_already_setup = True
 self.background_mobjects = []
 self.foreground_mobjects = []
 self.transformable_mobjects = []
 self.moving_vectors = []
 self.transformable_labels = []
 self.moving_mobjects = []

 self.background_plane = NumberPlane(**self.background_plane_kw_args)

 if self.show_coordinates:
 self.background_plane.add_coordinates()
 if self.include_background_plane:
 self.add_background_mobject(self.background_plane)
 if self.include_foreground_plane:
 self.plane = NumberPlane(**self.foreground_plane_kw_args)
 self.add_transformable_mobject(self.plane)
 if self.show_basis_vectors:
 self.basis_vectors = self.get_basis_vectors(
 i_hat_color=self.i_hat_color,
 j_hat_color=self.j_hat_color,
)
 self.moving_vectors += list(self.basis_vectors)
 self.i_hat, self.j_hat = self.basis_vectors
 self.add(self.basis_vectors)

def add_special_mobjects(self, mob_list: list, *mobs_to_add: Mobject):
 """
 Adds mobjects to a separate list that can be tracked,
 if these mobjects have some extra importance.

 Parameters

 mob_list
 The special list to which you want to add
 these mobjects.

 *mobs_to_add
 The mobjects to add.

 """
 for mobobject in mobs_to_add:
 if mobobject not in mob_list:
 mob_list.append(mobobject)
 self.add(mobobject)

def add_background_mobject(self, *mobjects: Mobject):
 """
 Adds the mobjects to the special list
 self.background_mobjects.

 Parameters

 *mobjects
 The mobjects to add to the list.

 """
 self.add_special_mobjects(self.background_mobjects, *mobjects)

TODO, this conflicts with Scene.add_fore
def add_foreground_mobject(self, *mobjects: Mobject):
 """
 """

```

```

 Adds the mobjects to the special list
 self.foreground_mobjects.

 Parameters

 *mobjects
 The mobjects to add to the list
 """
 self.add_special_mobjects(self.foreground_mobjects, *mobjects)

def add_transformable_mobject(self, *mobjects: Mobject):
 """
 Adds the mobjects to the special list
 self.transformable_mobjects.

 Parameters

 *mobjects
 The mobjects to add to the list.
 """
 self.add_special_mobjects(self.transformable_mobjects, *mobjects)

def add_moving_mobject(
 self, mobject: Mobject, target_mobject: Mobject | None = None
):
 """
 Adds the mobject to the special list
 self.moving_mobject, and adds a property
 to the mobject called mobject.target, which
 keeps track of what the mobject will move to
 or become etc.

 Parameters

 mobject
 The mobjects to add to the list

 target_mobject
 What the moving_mobject goes to, etc.
 """
 mobject.target = target_mobject
 self.add_special_mobjects(self.moving_mobjects, mobject)

def get_unit_square(
 self, color: str = YELLOW, opacity: float = 0.3, stroke_width: float = 3
):
 """
 Returns a unit square for the current NumberPlane.

 Parameters

 color
 The string of the hex color code of the color wanted.

 opacity
 The opacity of the square

 stroke_width
 The stroke_width in pixels of the border of the square

 Returns

 Square
 """
 square = self.square = Rectangle(
 color=color,
 width=self.plane.get_x_unit_size(),
 height=self.plane.get_y_unit_size(),
 stroke_color=color,
 stroke_width=stroke_width,
 fill_color=color,
 fill_opacity=opacity,
)
 square.move_to(self.plane.coords_to_point(0, 0), DL)
 return square

def add_unit_square(self, animate: bool = False, **kwargs):
 """
 Adds a unit square to the scene via
 self.get_unit_square.

 Parameters

 animate
 Whether or not to animate the addition
 with DrawBorderThenFill.
 **kwargs

```

```

Any valid keyword arguments of
self.get_unit_square()

>Returns

Square
 The unit square.
"""
square = self.get_unit_square(**kwargs)
if animate:
 self.play(
 DrawBorderThenFill(square),
 Animation(Group(*self.moving_vectors)),
)
self.add_transformable_mobject(square)
self.bring_to_front(*self.moving_vectors)
self.square = square
return self

def add_vector(
 self, vector: Arrow | list | tuple | np.ndarray, color: str = YELLOW, **kwargs
):
 """
 Adds a vector to the scene, and puts it in the special
 list self.moving_vectors.

 Parameters

 vector
 It can be a pre-made graphical vector, or the
 coordinates of one.

 color
 The string of the hex color of the vector.
 This is only taken into consideration if
 'vector' is not an Arrow. Defaults to YELLOW.

 **kwargs
 Any valid keyword argument of VectorScene.add_vector.

>Returns

Arrow
 The arrow representing the vector.
"""
vector = super().add_vector(vector, color=color, **kwargs)
self.moving_vectors.append(vector)
return vector

def write_vector_coordinates(self, vector: Arrow, **kwargs):
 """
 Returns a column matrix indicating the vector coordinates,
 after writing them to the screen, and adding them to the
 special list self.foreground_mobjects

 Parameters

 vector
 The arrow representing the vector.

 **kwargs
 Any valid keyword arguments of VectorScene.write_vector_coordinates

>Returns

Matrix
 The column matrix representing the vector.
"""
coords = super().write_vector_coordinates(vector, **kwargs)
self.add_foreground_mobject(coords)
return coords

def add_transformable_label(
 self,
 vector: Vector,
 label: MathTex | str,
 transformation_name: str | MathTex = "L",
 new_label: str | MathTex | None = None,
 **kwargs,
):
 """
 Method for creating, and animating the addition of
 a transformable label for the vector.

 Parameters

 vector

```

```

 The vector for which the label must be added.

label
 The MathTex/string of the label.

transformation_name
 The name to give the transformation as a label.

new_label
 What the label should display after a Linear Transformation

**kwargs
 Any valid keyword argument of get_vector_label

Returns

:class:`~.MathTex`
 The MathTex of the label.
"""

label_mob = self.label_vector(vector, label, **kwargs)
if new_label:
 label_mob.target_text = new_label
else:
 label_mob.target_text = "{}({})".format(
 transformation_name,
 label_mob.get_tex_string(),
)
label_mob.vector = vector
label_mob.kwargs = kwargs
if "animate" in label_mob.kwargs:
 label_mob.kwargs.pop("animate")
self.transformable_labels.append(label_mob)
return label_mob

def add_title(
 self,
 title: str | MathTex | Tex,
 scale_factor: float = 1.5,
 animate: bool = False,
) :
 """
 Adds a title, after scaling it, adding a background rectangle,
 moving it to the top and adding it to foreground_mobjects adding
 it as a local variable of self. Returns the Scene.

 Parameters

 title
 What the title should be.

 scale_factor
 How much the title should be scaled by.

 animate
 Whether or not to animate the addition.

 Returns

 LinearTransformationScene
 The scene with the title added to it.
 """

if not isinstance(title, (Mobject, OpenGLMobject)):
 title = Tex(title).scale(scale_factor)
title.to_edge(UP)
title.add_background_rectangle()
if animate:
 self.play(Write(title))
self.add_foreground_mobject(title)
self.title = title
return self

def get_matrix_transformation(self, matrix: np.ndarray | list | tuple):
 """
 Returns a function corresponding to the linear
 transformation represented by the matrix passed.

 Parameters

 matrix
 The matrix.
 """

 return self.get_transposed_matrix_transformation(np.array(matrix).T)

def get_transposed_matrix_transformation(
 self, transposed_matrix: np.ndarray | list | tuple
):
 """

```

```

 Returns a function corresponding to the linear
 transformation represented by the transposed
 matrix passed.

 Parameters

 transposed_matrix
 The matrix.
 """
 transposed_matrix = np.array(transposed_matrix)
 if transposed_matrix.shape == (2, 2):
 new_matrix = np.identity(3)
 new_matrix[:2, :2] = transposed_matrix
 transposed_matrix = new_matrix
 elif transposed_matrix.shape != (3, 3):
 raise ValueError("Matrix has bad dimensions")
 return lambda point: np.dot(point, transposed_matrix)

def get_piece_movement(self, pieces: list | tuple | np.ndarray):
 """
 This method returns an animation that moves an arbitrary
 mobject in "pieces" to its corresponding .target value.
 If self.leave_ghost_vectors is True, ghosts of the original
 positions/mobjects are left on screen

 Parameters

 pieces
 The pieces for which the movement must be shown.

 Returns

 Animation
 The animation of the movement.
 """
 start = VGroup(*pieces)
 target = VGroup(*(mob.target for mob in pieces))
 if self.leave_ghost_vectors:
 self.add(start.copy().fade(0.7))
 return Transform(start, target, lag_ratio=0)

def get_moving_mobject_movement(self, func: Callable[[np.ndarray], np.ndarray]):
 """
 This method returns an animation that moves a mobject
 in "self.moving_mobjects" to its corresponding .target value.
 func is a function that determines where the .target goes.

 Parameters

 func
 The function that determines where the .target of
 the moving mobject goes.

 Returns

 Animation
 The animation of the movement.
 """
 for m in self.moving_mobjects:
 if m.target is None:
 m.target = m.copy()
 target_point = func(m.get_center())
 m.target.move_to(target_point)
 return self.get_piece_movement(self.moving_mobjects)

def get_vector_movement(self, func: Callable[[np.ndarray], np.ndarray]):
 """
 This method returns an animation that moves a mobject
 in "self.moving_vectors" to its corresponding .target value.
 func is a function that determines where the .target goes.

 Parameters

 func
 The function that determines where the .target of
 the moving mobject goes.

 Returns

 Animation
 The animation of the movement.
 """
 for v in self.moving_vectors:
 v.target = Vector(func(v.get_end()), color=v.get_color())
 norm = np.linalg.norm(v.target.get_end())

```

```

 if norm < 0.1:
 v.target.get_tip().scale(norm)
 return self.get_piece_movement(self.moving_vectors)

def get_transformable_label_movement(self):
 """
 This method returns an animation that moves all labels
 in "self.transformable_labels" to its corresponding .target .

 Returns

 Animation
 The animation of the movement.
 """
 for label in self.transformable_labels:
 label.target = self.get_vector_label(
 label.vector.target, label.target_text, **label.kwargs
)
 return self.get_piece_movement(self.transformable_labels)

def apply_matrix(self, matrix: np.ndarray | list | tuple, **kwargs):
 """
 Applies the transformation represented by the
 given matrix to the number plane, and each vector/similar
 mobject on it.

 Parameters

 matrix
 The matrix.
 **kwargs
 Any valid keyword argument of self.apply_transposed_matrix()
 """
 self.apply_transposed_matrix(np.array(matrix).T, **kwargs)

def apply_inverse(self, matrix: np.ndarray | list | tuple, **kwargs):
 """
 This method applies the linear transformation
 represented by the inverse of the passed matrix
 to the number plane, and each vector/similar mobject on it.

 Parameters

 matrix
 The matrix whose inverse is to be applied.
 **kwargs
 Any valid keyword argument of self.apply_matrix()
 """
 self.apply_matrix(np.linalg.inv(matrix), **kwargs)

def apply_transposed_matrix(
 self, transposed_matrix: np.ndarray | list | tuple, **kwargs
):
 """
 Applies the transformation represented by the
 given transposed matrix to the number plane,
 and each vector/similar mobject on it.

 Parameters

 transposed_matrix
 The matrix.
 **kwargs
 Any valid keyword argument of self.apply_function()
 """
 func = self.get_transposed_matrix_transformation(transposed_matrix)
 if "path_arc" not in kwargs:
 net_rotation = np.mean(
 [angle_of_vector(func(RIGHT)), angle_of_vector(func(UP)) - np.pi / 2],
)
 kwargs["path_arc"] = net_rotation
 self.apply_function(func, **kwargs)

def apply_inverse_transpose(self, t_matrix: np.ndarray | list | tuple, **kwargs):
 """
 Applies the inverse of the transformation represented
 by the given transposed matrix to the number plane and each
 vector/similar mobject on it.

 Parameters

 t_matrix
 The matrix.
 **kwargs
 Any valid keyword argument of self.apply_transposed_matrix()
 """
 t_inv = np.linalg.inv(np.array(t_matrix).T).T

```

```

 self.apply_transposed_matrix(t_inv, **kwargs)

def apply_nonlinear_transformation(
 self, function: Callable[[np.ndarray], np.ndarray], **kwargs
):
 """
 Applies the non-linear transformation represented
 by the given function to the number plane and each
 vector/similar mobject on it.

 Parameters

 function
 The function.
 **kwargs
 Any valid keyword argument of self.apply_function()
 """
 self.plane.prepare_for_nonlinear_transform()
 self.apply_function(function, **kwargs)

def apply_function(
 self,
 function: Callable[[np.ndarray], np.ndarray],
 added_anims: list = [],
 **kwargs,
):
 """
 Applies the given function to each of the mobjects in
 self.transformable_mobjects, and plays the animation showing
 this.

 Parameters

 function
 The function that affects each point
 of each mobject in self.transformable_mobjects.

 added_anims
 Any other animations that need to be played
 simultaneously with this.

 **kwargs
 Any valid keyword argument of a self.play() call.
 """
 if "run_time" not in kwargs:
 kwargs["run_time"] = 3
 anims = (
 [
 ApplyPointwiseFunction(function, t_mob)
 for t_mob in self.transformable_mobjects
]
 +
 [
 self.get_vector_movement(function),
 self.get_transformable_label_movement(),
 self.get_moving_mobject_movement(function),
]
 +
 [Animation(f_mob) for f_mob in self.foreground_mobjects]
 +
 added_anims
)
 self.play(*anims, **kwargs)

```