from os import system

from nea\_game.config import NeaGameConfig

if \_\_name\_\_ == "\_\_main\_\_":

    config = NeaGameConfig()

    if config.debug:

        system(f"python app.py --fname {config.debug\_file}")

    else:

        system("python app.py")

def lerp(a: float, b: float, lerp\_amount: float) -> float:

    """Linearly interpolates between two points.

    Interpolates between the points a and b by the interpolant lerp\_amount.

    The parameter lerp\_amount is clamped to the range [0, 1].

    This is most commonly used to find a point some fraction of the way along a line between two endpoints

        Args:

            a (float): Endpoint 1

            b (float): Endpoint 2

            lerp\_amount (float): The interpolent

        Returns:

            float: The interpolated value

    """

    return (a \* (1 - lerp\_amount)) + (b \* lerp\_amount)

"""Returns the sign of flt. Return value is 1 when flt is positive or zero, -1 when flt is negative.

"""

def sign(flt: float) -> float:

    """Returns the sign of flt.

        Return value is 1 when flt is positive or zero, -1 when flt is negative.

    Args:

        flt (float): The value to determine the sign of

    Returns:

        float: The sign of flt

    """

    return 1 if flt >= 0 else -1

"""Defines a thresholod and a funtion to approximate floats close to zero

"""

from math import isclose

from functools import partial

THRESHOLD = 1e-1

is\_close\_to\_zero = partial(isclose, b=0, abs\_tol=THRESHOLD)

def near\_zero(flt: float) -> float:

    """If a float is close to zero then zero is returned othewise the flaot itself is returned

    Args:

        flt (float): The float to be approximated

    Returns:

        float: The approximated value of the float

    """

    if is\_close\_to\_zero(flt):

        return 0

    return flt

"""Provides a Vector class in the 2-Dimensional plane with the associated methods for vectors"""

from \_\_future\_\_ import annotations

from typing import NamedTuple

from math import sqrt

from nea\_game.calc.near\_zero import near\_zero

class Vector2D(NamedTuple):

    """Creates a 2 dimensional vector"""

    x: float

    y: float

    def \_\_repr\_\_(self) -> str:

        """Returns a string representation of the vector displaying the both the x and y components

        Returns:

            String: The resulting representation of the vector

        """

        return f"Vector2D: ({self.x, self.y})"

    def \_\_add\_\_(self, other: Vector2D) -> Vector2D:

        """Calculates the sum of 2 vectors

        Args:

            other (Vector2D): The other vector to add

        Returns:

            Vector2D: The sum of the 2 vectors

        """

        return Vector2D(self.x + other.x, self.y + other.y)

    def near\_zero(self) -> Vector2D:

        return Vector2D(near\_zero(self.x), near\_zero(self.y))

    def magnitude(self) -> float:

        """Returns the magnitude of the vector

        Returns:

            float: The magnitude of the vector

        """

        return sqrt(self.x \* self.x + self.y \* self.y)

    def normalise(self) -> Vector2D:

        """Returns a vector with the same direction but a magnitude of 1

        Returns:

            Vector2D: The vector with the same direction but a magnitude of 1

        """

        return Vector2D(self.x / self.magnitude(), self.y / self.magnitude())

    def scale(self, scale\_factor: float) -> Vector2D:

        """Scales the vector by a given scale factor

        Args:

            scale\_factor (float): The amount that the vector is scaled by

        Returns:

            Vector2D: The scaled vector

        """

        return Vector2D(self.x \* scale\_factor, self.y \* scale\_factor)

    def scale\_to\_length(self, length: float) -> Vector2D:

        """Scales the vector to a given length

        Args:

            length (float): The length that the vector is scaled to

        Raises:

            ValueError: Raises a ValueError when the magnitude of the vector is 0 as

            that can't be scaled to a given length

        Returns:

            Vector2D: The vector scaled to the given length

        """

        if self.magnitude() == 0:

            raise ValueError("The Magnitude of the vector must be greater than zero")

        return self.normalise().scale(length)

    def dot(self, other: Vector2D) -> float:

        """Calculates the dot product with the other vector

        Args:

            other (Vector2D): The other vector that is used to calculate the dot product

        Returns:

            Float: The resulting scalar from performing the dot product

        """

        return self.x \* other.x + self.y \* other.y

    def cross(self, other: Vector2D) -> float:

        """Calculates the cross product with the other vector

        Args:

            other (Vector2D): The other vector that is used to calculate the cross product

        Returns:

            Float: The resulting scalar from performing the cross product

        """

        return (self.x \* other.y) - (self.y \* other.x)

from pygame.event import Event

import pygame

from nea\_game.components.base\_component import BaseComponent

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.calc.vector2d import Vector2D

class Input(BaseComponent):

    """Provides an interface for accessing the inputs of a player given actions"""

    actions: dict[PlayerActionSpace, int]

    def \_\_init\_\_(

        self, action\_space: type[PlayerActionSpace], action\_bindings: list[int]

    ):

        """Summary

        Args:

            action\_space (type[PlayerActionSpace]): An Enum for each possible action the player can make

            action\_bindings (list[int]): The binding for each corresponding action in action\_space

        """

        self.action\_space = action\_space

        self.actions = dict(zip(list(self.action\_space), action\_bindings))

        self.actions\_performed\_on\_current\_frame = {

            action: False for action in list(self.action\_space)

        }

    def get\_axis\_raw(self) -> Vector2D:

        """Calculates the vector of the directional inputs based on the bindings of the actions UP, DOWN, LEFT, RIGHT. The value for both axis will be either -1, 0 or 1.

        Returns:

            Vector2D: The vector of the directional inputs"""

        horizontal = (

            -1

            if pygame.key.get\_pressed()[self.actions[self.action\_space.LEFT]]

            else 1

            \* (

                pygame.key.get\_pressed()[self.actions[self.action\_space.LEFT]]

                ^ pygame.key.get\_pressed()[self.actions[self.action\_space.RIGHT]]

            )

        )

        vertical = (

            -1

            if pygame.key.get\_pressed()[self.actions[self.action\_space.DOWN]]

            else 1

            \* (

                pygame.key.get\_pressed()[self.actions[self.action\_space.DOWN]]

                ^ pygame.key.get\_pressed()[self.actions[self.action\_space.UP]]

            )

        )

        return Vector2D(horizontal, vertical)

    def update\_actions\_performed\_on\_current\_frame(self, events: list[Event]):

        """Updates the dictionary corresponding to the actions performed on the current frame"""

        self.actions\_performed\_on\_current\_frame = {

            action: False for action in self.action\_space

        }

        for event in events:

            if event.type == pygame.KEYDOWN:

                for action, binding in self.actions.items():

                    if event.key == binding:

                        self.actions\_performed\_on\_current\_frame[action] = True

    def get\_action\_down(self, action: PlayerActionSpace) -> bool:

        """Returns true during the frame the user starts pressing down the key identified by the action action enum parameter.

        Args:

            action (Enum): The action that is being checked

        Returns:

            bool: Whether that action was performed that frame

        """

        return self.actions\_performed\_on\_current\_frame[action]

import pygame

class StaticRenderer:

    def \_\_init\_\_(self, static\_frame: pygame.Surface):

        """Creates a StaticRenderer component for a static object

        Args:

            static\_frame (pygame.Surface): The static frame to be rendered

        """

        self.static\_frame = static\_frame

    def render\_entity(self, surface: pygame.Surface, x: float, y: float):

        """Renders the static frame onto the surface at the given position

        Args:

            surface (pygame.Surface): The surface that the frame is rendered onto

            x (int): The x position that the static frame is rendered onto

            y (int): The y position that the static frame is rendered onto

        """

        surface.blit(self.static\_frame, (x, y))

class AnimatedRenderer:

    def \_\_init\_\_(

        self, frames: dict[str, list[pygame.Surface]], current\_frame\_index: int = 0

    ):

        """Creates an AnimatedRenderer component for an animated object

        Args:

            frames (list[pygame.Surface]): The list of each possible frame for the animated object

            current\_frame\_index (int, optional): If specified the starting frame will be the frame specified at the given index, otherwise the first one is selected

        """

        self.frames = frames

        self.current\_frame\_index = current\_frame\_index

    def render\_entity(

        self, state\_name: str, flip\_x: bool, surface: pygame.Surface, x: float, y: float

    ):

        """Renders the current frame onto the surface at the given position

        Args:

            state\_name (str): The current state of the entity

            surface (pygame.Surface): The surface that the frame is rendered onto

            x (int): The x position that the frame is rendered onto

            y (int): The y position that the frame is rendered onto

        """

        surface.blit(

            pygame.transform.flip(

                self.frames[state\_name][self.current\_frame\_index], flip\_x, False

            ),

            (x, y),

        )

    def set\_current\_frame(self, new\_frame\_index: int):

        """Sets the index of the current frame to the index specified

        Args:

            new\_frame\_index (int): The new index of the current frame

        Raises:

            IndexError: Raises an IndexError if the given index is outside of the range

        """

        if new\_frame\_index < 0 or new\_frame\_index >= len(self.frames):

            raise IndexError(

                f"The given index: {new\_frame\_index} is outside of the range 0 <= new\_frame\_index <= {len(self.frames)})"

            )

        self.current\_frame\_index = new\_frame\_index

from enum import Enum, auto

from nea\_game.calc.vector2d import Vector2D

class ForceMode(Enum):

    """Provides a container for the valid ways for a force to be applied to a rigid body

    Attributes:

        FORCE (TYPE): Add a continuous force to the rigidbody, using its mass

        IMPULSE (TYPE): Add an instant force impulse to the rigidbody, using its mass.

    """

    FORCE = auto()

    IMPULSE = auto()

    ACCELERATION = auto()

class RigidBody2D:

    """Creates a 2D rigid body that forces can be applied to"""

    def \_\_init\_\_(self, mass: float, gravity\_scale: float, internal\_fps: int) -> None:

        """Creates a 2D rigid body

        Args:

            mass (float): The mass of the body in kg

            gravity\_scale (float): The degree to which the body is affected by gravity

        """

        self.mass = mass

        self.gravity\_scale = gravity\_scale

        self.internal\_fps = internal\_fps

        self.velocity = Vector2D(0, 0)

    def add\_force(

        self, force: Vector2D, dt: float = 0, force\_mode: ForceMode = ForceMode.FORCE

    ) -> None:

        """Adds a force to the rigid body

        Args:

            force (Vector2D): The unscaled force that will act on the body

            dt (float): The time difference between the previous frame that was drawn and the current frame

            force\_mode (ForceModes, optional): The mode that the force can be applied

        Raises:

            ValueError: A ValueError is raised when the given force\_mode is not a valid force\_type

        """

        dt \*= self.internal\_fps

        match force\_mode:

            case ForceMode.FORCE:

                self.velocity += force.scale(dt / self.mass)

            case ForceMode.IMPULSE:

                self.velocity += force.scale(1 / self.mass)

            case ForceMode.ACCELERATION:

                self.velocity += force.scale(dt)

            case \_:

                raise ValueError(

                    f"The given force\_type: {force\_mode} is not in {[member.value for member in ForceMode]}"

                )

        self.velocity = self.velocity.near\_zero()

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

class BaseEntity:

    x: float

    y: float

    level\_data: list[LevelTile]

    def \_\_init\_\_(self, x: float, y: float, level\_data: list[LevelTile]) -> None:

        """Provides a template for all entities

        Args:

            x (float): The x position of the entity

            y (float): The y position of the entity

        """

        self.x = x

        self.y = y

        self.level\_data = level\_data

    def update(self, dt: float) -> None:

        """Called each frame in order to update the entity"""

from pygame import Rect

class Camera:

    def \_\_init\_\_(

        self,

        height: int,

        width: int,

        surface\_height: int,

        surface\_width: int,

    ):

        self.height = height

        self.width = width

        self.surface\_height = surface\_height

        self.surface\_width = surface\_width

        self.scroll\_x: int = 0

        self.scroll\_y: int = 0

        self.trauma: float = 0

    def update(self, target\_rect: Rect):

        """Updates the scroll values of the camera to focus on the target

        Args:

            target\_rect (Rect): The rect for a given target

        """

        self.scroll\_x += int(

            (

                target\_rect.x

                - self.scroll\_x

                - (self.surface\_width + target\_rect.width) // 2

            )

            \* 0.2

        )

        self.scroll\_y += int(

            (

                target\_rect.y

                - self.scroll\_y

                - (self.surface\_height + target\_rect.height) // 2

            )

            \* 0.2

        )

        self.scroll\_x = min(max(0, self.scroll\_x), self.width - self.surface\_width)

        self.scroll\_y = min(max(0, self.scroll\_y), self.height - self.surface\_height)

from time import perf\_counter

from pygame.time import Clock

class Engine:

    """

    Provides an interface to access the time between the last frame

    """

    def \_\_init\_\_(self, fps: int):

        self.clock = Clock()

        self.t1 = perf\_counter()

        self.fps = fps

    def update(self):

        self.clock.tick(self.fps)

        self.dt = perf\_counter() - self.t1

        self.t1 = perf\_counter()

from \_\_future\_\_ import annotations

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.event import Event

from pygame import Rect, Surface

from nea\_game.game.camera import Camera

from nea\_game.gui.window import Window

from nea\_game.menu.background\_layer import BackgroundLayer

from nea\_game.ldtk\_world\_loader.world import World

from nea\_game.player.player import Player

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class Game(Window):

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        world\_number: int,

        level\_number: int,

        background\_image\_layers\_path: Path,

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.config = self.parent.config

        self.world\_number = world\_number

        self.level\_number = level\_number

        self.world = World(self.world\_number, self.config.directories["worlds"])

        self.background\_layers = [

            BackgroundLayer(pygame.image.load(background\_image\_layers\_path / filename))

            for filename in sorted(listdir(background\_image\_layers\_path))

            if filename.endswith(".png") and filename not in ["-1.png", "2.png"]

        ]

        self.player = Player(

            self.config.directories["player"],

            self.world.levels[0].level\_data,

            self.config.key\_bindings,

            self.config.internal\_fps,

            17,

            160,

        )

        self.camera = Camera(

            self.world.levels[0].height,

            self.world.levels[0].width,

            self.display\_surface.get\_height(),

            self.display\_surface.get\_width(),

        )

    def event\_handler(self, events: list[Event]):

        self.player.event\_handler(events)

    def update(self, dt: float):

        self.player.update(dt)

        #print(self.player.state\_machine.current\_state.state\_name)

        self.camera.update(self.player.rect)

    def draw(self):

        self.display\_surface.fill((0, 0, 0))

        self.display\_surface.blit(self.background\_layers[0].image, (0, 0))

        for background\_layer in self.background\_layers[1:2]:

            self.display\_surface.blit(background\_layer.get\_new\_sub\_image(), (0, 0))

        for tile in self.world.levels[0].level\_data:

            self.display\_surface.blit(

                tile.image,

                (

                    tile.rect.left - self.camera.scroll\_x,

                    tile.rect.top - self.camera.scroll\_y,

                ),

            )

        self.player.renderer.render\_entity(

            "idle",

            self.player.direction == -1,

            self.display\_surface,

            self.player.rect.x - self.camera.scroll\_x,

            self.player.rect.y - self.camera.scroll\_y,

        )

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

from os.path import join

from pathlib import Path

from pygame.image import load

from pygame import Mask

from pygame import Rect

from pygame import Surface

from pygame.mask import from\_surface

class Button:

    passive\_image: Surface

    active\_image: Surface

    on\_click\_image: Surface

    current\_image: Surface

    rect: Rect

    mask: Mask

    can\_be\_clicked: bool

    clicked: bool

    click\_delay: float

    click\_timer: float

    def \_\_init\_\_(self, path: Path):

        self.passive\_image = load(join(path, "0.png")).convert\_alpha()

        self.active\_image = load(join(path, "1.png")).convert\_alpha()

        self.on\_click\_image = load(join(path, "2.png")).convert\_alpha()

        self.current\_image = self.passive\_image

        self.rect = self.passive\_image.get\_rect()

        self.mask = from\_surface(self.passive\_image)

        self.rect.topleft = 0, 0

        self.can\_be\_clicked = True

        self.clicked = False

        self.click\_delay = 0.08

        self.click\_timer = -1

    def update(self, mouse\_pos: tuple[int, int], mouse\_clicked: bool, dt: float):

        """Updates the current\_image attribute and the clicked flag

        Args:

            mouse\_pos (tuple[int, int]): The position of the mouse on the screen

            mouse\_clicked (bool): A boolean value stating whether the mouse is clicked or not on the current frame

        """

        mouse\_pos\_x, mouse\_pos\_y = mouse\_pos

        mouse\_pos\_in\_mask = mouse\_pos\_x - self.rect.x, mouse\_pos\_y - self.rect.y

        self.clicked = False

        if self.rect.collidepoint(mouse\_pos) and self.mask.get\_at(mouse\_pos\_in\_mask):

            if mouse\_clicked and self.can\_be\_clicked:

                self.current\_image = self.on\_click\_image

                if self.click\_timer == -1:

                    self.click\_timer = 0

            else:

                self.current\_image = self.active\_image

        else:

            self.current\_image = self.passive\_image

        if self.click\_timer > self.click\_delay:

            self.clicked = True

            self.click\_timer = -1

        if self.click\_timer >= 0:

            self.click\_timer += dt

    def center\_on\_x\_axis(self, x\_axis\_width: int):

        self.rect.x = (x\_axis\_width - self.rect.width) // 2

import pygame

from pygame.event import Event

from pygame import Surface

from nea\_game.gui.window import Window

from nea\_game.game.engine import Engine

class Root:

    screen: Surface

    active\_window: Window

    windows: dict[str, Window]

    engine: Engine

    def \_\_init\_\_(

        self,

        screen\_resolution: tuple[int, int],

        display\_surafce\_resolution: tuple[int, int],

        fps: int,

    ):

        pygame.init()

        self.screen = pygame.display.set\_mode(screen\_resolution)

        self.display\_surface = Surface(display\_surafce\_resolution)

        self.engine = Engine(fps)

        self.windows = {}

    def show\_window(self, window: str):

        self.active\_window = self.windows[window]

    def update(self):

        self.engine.update()

        self.active\_window.update(self.engine.dt)

        self.active\_window.draw()

from pathlib import Path

from pygame.image import load

from pygame import Mask

from pygame import Rect

from pygame import Surface

from pygame.mask import from\_surface

class Title:

    image: Surface

    rect: Rect

    mask: Mask

    def \_\_init\_\_(self, title\_image\_path: Path):

        self.image = load(title\_image\_path).convert\_alpha()

        self.rect = self.image.get\_rect()

        self.mask = from\_surface(self.image)

        self.rect.topleft = 0, 0

    def center\_on\_x\_axis(self, x\_axis\_width: int):

        self.rect.x = (x\_axis\_width - self.rect.width) // 2

from pygame.event import Event

from pygame import Surface

class Window:

    screen: Surface

    display\_surface: Surface

    def \_\_init\_\_(self, screen: Surface, display\_surface: Surface):

        self.screen = screen

        self.display\_surface = display\_surface

        self.scale\_factor: int = (

            self.screen.get\_width() // self.display\_surface.get\_width()

        )

        if (

            self.scale\_factor

            != self.screen.get\_height() // self.display\_surface.get\_height()

        ):

            raise ValueError("Display surface and screen must be proportional")

    def event\_handler(self, events: list[Event]):

        """Passes down the event to the event\_handler for any relvant operations

        Args:

            events (list[Event]): A list of pygame events

        """

    def update(self, dt: float):

        """Updates window"""

    def draw(self):

        """Draws window"""

"""An enum for the different collision types a tile can have

"""

from enum import Enum, auto

class CollisionType(Enum):

    """An enum for the collision types a tile can have"""

    WALL = auto()

    SPIKE = auto()

    PLATFORM = auto()

from typing import Self

from pygame import Surface

from nea\_game.ldtk\_world\_loader.tileset\_tile import TilesetTile

class LevelTile(TilesetTile):

    def \_\_init\_\_(

        self,

        identifier: int,

        image: Surface,

        collision\_type: int,

        size: int,

        px: tuple[int, int],

    ):

        super().\_\_init\_\_(identifier, image, collision\_type, size)

        self.rect.topleft = px

    @classmethod

    def from\_tileset\_tile(cls, tileset\_tile: TilesetTile, px: tuple[int, int]) -> Self:

        return cls(

            tileset\_tile.identifier,

            tileset\_tile.image,

            tileset\_tile.collision\_type,

            tileset\_tile.rect.width,

            px,

        )

from typing import Any

from nea\_game.config import NeaGameConfig

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

from nea\_game.ldtk\_world\_loader.tileset import Tileset

from nea\_game.player.player import Player

class Level:

    height: int

    width: int

    def \_\_init\_\_(self, data: dict[str, Any], tileset: Tileset):

        self.data = data

        self.tileset = tileset

        self.height = self.data["pxHei"]

        self.width = self.data["pxWid"]

        self.level\_data = self.generate\_level\_data()

    def generate\_level\_data(self):

        level\_data: list[LevelTile] = []

        for layer in self.data["layerInstances"]:

            if layer["\_\_identifier"] == "AutoTiles":

                for tile in layer["autoLayerTiles"]:

                    level\_data.append(

                        LevelTile.from\_tileset\_tile(

                            self.tileset.tiles[tile["t"]], tile["px"]

                        )

                    )

            if layer["\_\_identifier"] == "Tiles":

                for tile in layer["gridTiles"]:

                    level\_data.append(

                        LevelTile.from\_tileset\_tile(

                            self.tileset.tiles[tile["t"]], tile["px"]

                        )

                    )

        return level\_data

from pygame import Rect, Surface

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

class TilesetTile:

    def \_\_init\_\_(

        self, identifier: int, image: Surface, collision\_type: CollisionType, size: int

    ):

        self.identifier = identifier

        self.image = image

        self.collision\_type = collision\_type

        self.rect = Rect(0, 0, size, size)

from pathlib import Path

from typing import Any

from pygame.image import load

from pygame import Rect

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

from nea\_game.ldtk\_world\_loader.tileset\_tile import TilesetTile

class Tileset:

    grid\_height: int

    grid\_width: int

    grid\_size: int

    spacing: int

    padding: int

    collision\_types: list[dict[str, Any]]

    tiles: dict[int, TilesetTile]

    def \_\_init\_\_(self, data: dict[str, Any], world\_path: Path):

        self.grid\_height = data["\_\_cHei"]

        self.grid\_width = data["\_\_cWid"]

        self.grid\_size = data["tileGridSize"]

        self.spacing = data["spacing"]

        self.padding = data["padding"]

        self.image = load(world\_path / data["relPath"])

        self.collision\_types: list[dict[str, Any]] = data["enumTags"]

        tiles: dict[int, str] = {}

        for collision\_type in self.collision\_types:

            for tile\_id in collision\_type["tileIds"]:

                tiles[tile\_id] = collision\_type["enumValueId"]

        self.tiles = {}

        for tile\_id, collision\_type in tiles.items():

            grid\_x = tile\_id - (self.grid\_width \* (tile\_id // self.grid\_width))

            pixel\_x = self.padding + (grid\_x \* (self.grid\_size + self.spacing))

            grid\_y = tile\_id // self.grid\_width

            pixel\_y = self.padding + (grid\_y \* (self.grid\_size + self.spacing))

            handle\_image = self.image.copy()

            clip\_rect = Rect(pixel\_x, pixel\_y, self.grid\_size, self.grid\_size)

            handle\_image.set\_clip(clip\_rect)

            tile\_image = self.image.subsurface(handle\_image.get\_clip())

            self.tiles[tile\_id] = TilesetTile(

                tile\_id,

                tile\_image,

                getattr(CollisionType, collision\_type.upper()),

                self.grid\_size,

            )

from json import load

from pathlib import Path

from nea\_game.ldtk\_world\_loader.level import Level

from nea\_game.ldtk\_world\_loader.tileset import Tileset

class World:

    def \_\_init\_\_(self, world\_number: int, world\_directory: Path):

        with (world\_directory / f"{world\_number}.json").open() as world\_json:

            self.data = load(world\_json)

        self.tileset = Tileset(self.data["defs"]["tilesets"][0], world\_directory)

        self.levels: list[Level] = [

            Level(level\_data, self.tileset) for level\_data in self.data["levels"]

        ]

from pathlib import Path

import pygame

from pygame.event import Event

from pygame.image import load

from pygame.key import name as get\_key\_name

from pygame.mask import Mask, from\_surface

from pygame import Rect, Surface

class ActionButton:

    passive\_image: Surface

    active\_image: Surface

    current\_image: Surface

    rect: Rect

    mask: Mask

    key: int

    key\_images\_path: Path

    key\_image: Surface

    clicked: bool

    click\_time: float

    click\_timer: float

    def \_\_init\_\_(self, path: Path, key\_images\_path: Path, key: int):

        self.passive\_image = load(path / "0.png")

        self.active\_image = load(path / "1.png")

        self.current\_image = self.passive\_image

        self.rect = self.passive\_image.get\_rect()

        self.mask = from\_surface(self.passive\_image)

        self.rect.topleft = 0, 0

        self.key\_images\_path = key\_images\_path

        self.initial\_update\_key\_image(key)

        self.clicked = False

        self.click\_time = 5

        self.click\_timer = -1

    def update(

        self,

        mouse\_pos: tuple[int, int],

        mouse\_clicked: bool,

        dt: float,

        other\_binds: list[int] | None = None,

        event: Event | None = None,

    ):

        """Updates the current\_image attribute and the clicked flag

        Args:

            mouse\_pos (tuple[int, int]): The position of the mouse on the screen

            mouse\_clicked (bool): A boolean value stating whether the mouse is clicked or not on the current frame

        """

        mouse\_pos\_x, mouse\_pos\_y = mouse\_pos

        mouse\_pos\_in\_mask = mouse\_pos\_x - self.rect.x, mouse\_pos\_y - self.rect.y

        self.clicked = False

        if (

            self.rect.collidepoint(mouse\_pos)

            and self.mask.get\_at(mouse\_pos\_in\_mask)

            and mouse\_clicked

            or self.click\_timer >= 0

        ):

            self.current\_image = self.active\_image

            self.clicked = True

            self.click\_timer = max(0, self.click\_timer)

        else:

            self.current\_image = self.passive\_image

        if self.click\_timer >= 0:

            self.click\_timer += dt

            if event:

                if event.type == pygame.KEYDOWN:

                    if self.update\_key\_image(event.key, other\_binds):

                        self.click\_timer = -1

        if self.click\_timer >= self.click\_time:

            self.click\_timer = -1

    def initial\_update\_key\_image(self, key: int):

        key\_name = get\_key\_name(key).replace(" ", "\_")

        try:

            self.key\_image = load(self.key\_images\_path / f"{key\_name}-key.png")

        except FileNotFoundError:

            return False

        handle\_key\_image = self.key\_image.copy()

        clip\_rect = Rect(

            self.key\_image.get\_width() // 2,

            0,

            self.key\_image.get\_width() // 2,

            self.key\_image.get\_height(),

        )

        handle\_key\_image.set\_clip(clip\_rect)

        self.key = key

        self.key\_image = self.key\_image.subsurface(

            handle\_key\_image.get\_clip()

        ).convert\_alpha()

        return True

    def update\_key\_image(self, key: int, other\_binds: list[int] | None = None):

        if other\_binds is None:

            return False

        if key in other\_binds:

            return False

        key\_name = get\_key\_name(key).replace(" ", "\_")

        try:

            self.key\_image = load(self.key\_images\_path / f"{key\_name}-key.png")

        except FileNotFoundError:

            return False

        handle\_key\_image = self.key\_image.copy()

        clip\_rect = Rect(

            self.key\_image.get\_width() // 2,

            0,

            self.key\_image.get\_width() // 2,

            self.key\_image.get\_height(),

        )

        handle\_key\_image.set\_clip(clip\_rect)

        self.key = key

        self.key\_image = self.key\_image.subsurface(

            handle\_key\_image.get\_clip()

        ).convert\_alpha()

        return True

from math import sin

from time import perf\_counter

from random import uniform

from pygame import Rect, Surface

class BackgroundLayer:

    image: Surface

    sine\_scale\_factor: float

    sine\_stretch\_factor: float

    x\_scroll: float

    def \_\_init\_\_(self, image: Surface):

        self.image = image

        # self.sine\_scale\_factor = 0.01

        # self.sine\_stretch\_factor = pow(10, 20)

        # self.sine\_translation\_factor = 0.5

        self.sine\_scale\_factor = uniform(0.001, 0.1)

        self.sine\_stretch\_factor = pow(10, 20)

        self.sine\_translation\_factor = uniform(0.1, 0.7)

        self.x\_scroll = 0

    def get\_new\_sub\_image(self, x\_scroll: float = 0, y\_scroll: float = 0) -> Surface:

        """Generates a new scroll value to adjust which subsection of the image is returned

        Returns:

            Surface: The sub image according to the newly generated scroll value

        """

        self.x\_scroll += (

            self.sine\_scale\_factor \* abs(sin(perf\_counter() \* self.sine\_stretch\_factor))

            + self.sine\_translation\_factor

        ) + x\_scroll

        x\_scroll = int(self.x\_scroll) % (self.image.get\_width() // 2)

        handle\_image = self.image.copy()

        clip\_rect = Rect(

            x\_scroll, y\_scroll, self.image.get\_width() // 2, self.image.get\_height()

        )

        handle\_image.set\_clip(clip\_rect)

        return self.image.subsurface(handle\_image.get\_clip())

from \_\_future\_\_ import annotations

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.game.game import Game

from nea\_game.gui.button import Button

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class LevelSelection(Window):

    buttons: dict[str, Button]

    padding: tuple[int, int]

    spacing: tuple[int, int]

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display: Surface,

        button\_folder\_path: Path,

    ):

        super().\_\_init\_\_(screen, display)

        self.parent = parent

        self.padding = (30, 50)

        self.spacing = (59, 38)

        self.buttons = {}

        for button in listdir(button\_folder\_path):

            self.buttons[button] = Button(button\_folder\_path / button)

        for button\_name, button in zip(self.buttons, self.buttons.values()):

            if button\_name == "back":

                button.rect.bottomright = (383, 215)

            else:

                world\_num = int(button\_name[0])

                level\_num = int(button\_name[2])

                button.rect.topleft = (

                    self.padding[0] + (level\_num - 1) \* self.spacing[0],

                    self.padding[1] + (world\_num - 1) \* self.spacing[1],

                )

        for button, unlocked in zip(

            list(self.buttons.values())[:-1], self.parent.config.unlocked\_levels

        ):

            if not unlocked:

                button.active\_image = button.passive\_image

                button.on\_click\_image = button.passive\_image

    def update(self, dt: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, dt)

        if self.buttons["back"].clicked:

            self.parent.show\_window("main\_menu")

        if self.buttons["1-1"].clicked:

            window = Game(

                self.parent,

                self.screen,

                self.display\_surface,

                1,

                1,

                self.parent.config.directories["background"] / "sky\_mountain",

            )

            self.parent.windows["game"] = window

            self.parent.show\_window("game")

    def draw(self):

        self.display\_surface.fill((8, 169, 252))

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

from \_\_future\_\_ import annotations

from os import listdir, path

from pathlib import Path

from sys import exit as sys\_exit

import typing

import pygame

from pygame.image import load

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.menu.level\_selection import LevelSelection

from nea\_game.menu.settings import Settings

from nea\_game.menu.background\_layer import BackgroundLayer

from nea\_game.gui.button import Button

from nea\_game.gui.title import Title

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class MainMenu(Window):

    buttons: dict[str, Button]

    background\_layers: list[BackgroundLayer]

    splash\_screen\_opacity: int

    title: Title

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        title\_image\_path: Path,

        background\_image\_layers\_path: Path,

        button\_folder\_path: Path,

        background\_transparency\_percentage: float = 1,

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.buttons = {}

        self.title\_image = load(title\_image\_path).convert\_alpha()

        self.background\_layers = [

            BackgroundLayer(load(background\_image\_layers\_path / filename))

            for filename in sorted(listdir(background\_image\_layers\_path))

            if filename.endswith(".png") and filename != "-1.png"

        ]

        self.set\_background\_transparency\_percentage(background\_transparency\_percentage)

        self.title = Title(title\_image\_path)

        self.title.rect.y = 0

        self.title.center\_on\_x\_axis(self.display\_surface.get\_width())

        for button in listdir(button\_folder\_path):

            self.buttons[button] = Button(button\_folder\_path / button)

        self.buttons["play\_game"].rect.y = 115

        self.buttons["play\_game"].center\_on\_x\_axis(self.display\_surface.get\_width())

        self.buttons["settings"].rect.y = 150

        self.buttons["settings"].center\_on\_x\_axis(self.display\_surface.get\_width())

        self.buttons["exit"].rect.y = 185

        self.buttons["exit"].center\_on\_x\_axis(self.display\_surface.get\_width())

    def update(self, dt: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, dt)

        if self.buttons["play\_game"].clicked:

            window = LevelSelection(

                self.parent,

                self.screen,

                self.display\_surface,

                self.parent.config.directories["assets"] / "buttons/level\_selection",

            )

            self.parent.windows["play\_game"] = window

            self.parent.show\_window("play\_game")

        if self.buttons["settings"].clicked:

            window = Settings(

                self.parent,

                self.screen,

                self.display\_surface,

                self.parent.config.directories["assets"] / "actions",

                self.parent.config.key\_bindings,

            )

            self.parent.windows["settings"] = window

            self.parent.show\_window("settings")

        if self.buttons["exit"].clicked:

            pygame.quit()

            sys\_exit()

    def draw(self):

        self.display\_surface.fill((0, 0, 0))

        self.display\_surface.blit(self.background\_layers[0].image, (0, 0))

        self.display\_surface.blit(self.background\_layers[1].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[2].image, (0, 0))

        self.display\_surface.blit(self.background\_layers[3].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[4].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[5].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.title.image, self.title.rect.topleft)

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

    def set\_background\_transparency\_percentage(self, transparency\_percentage: float):

        """Sets the transparency for the background image as a number between 0 and 255 from a percenatge

        Args:

            transparency\_percentage (float): The percentage transparency for the back ground image (0= fully transparent, 1=fully opaque)

        Raises:

            ValueError: Raised when the transparency percentage is not between 0 and 1 inclusive

        """

        if transparency\_percentage < 0 or transparency\_percentage > 1:

            raise ValueError("Percentage must be between 0 and 1 inlcusive")

        for layer in self.background\_layers:

            layer.image.set\_alpha(int(255 \* transparency\_percentage))

from \_\_future\_\_ import annotations

from json import dump, load as load\_json

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.image import load

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.menu.action\_button import ActionButton

from nea\_game.gui.button import Button

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class Settings(Window):

    buttons: dict[str, Button]

    action\_buttons: dict[str, ActionButton]

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        action\_button\_folder\_path: Path,

        key\_bindings: list[int],

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.title\_image = load(

            action\_button\_folder\_path.parent / "settings\_title.png"

        ).convert\_alpha()

        self.text = load(

            action\_button\_folder\_path.parent / "settings\_text.png"

        ).convert\_alpha()

        self.buttons = {}

        self.action\_buttons = {}

        for button in listdir(action\_button\_folder\_path.parent / "buttons/settings"):

            self.buttons[button] = Button(

                (action\_button\_folder\_path.parent / "buttons/settings") / button

            )

        for action\_button, key\_binding in zip(

            listdir(action\_button\_folder\_path), key\_bindings

        ):

            self.action\_buttons[action\_button] = ActionButton(

                action\_button\_folder\_path / action\_button,

                action\_button\_folder\_path.parent / "keys",

                key\_binding,

            )

        self.buttons["back"].rect.bottomright = (383, 215)

        for index, action\_button in enumerate(self.action\_buttons.values()):

            action\_button.rect.topleft = (20, 20 + 40 \* index)

    def update(self, dt: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, dt)

        for action\_button in self.action\_buttons.values():

            if action\_button.click\_timer >= 0:

                other\_binds = [

                    other\_action\_button.key

                    for other\_action\_button in self.action\_buttons.values()

                    if other\_action\_button != action\_button

                ]

                for event in pygame.event.get():

                    action\_button.update(

                        scaled\_mouse\_pos, mouse\_clicked, 0, other\_binds, event

                    )

            action\_button.update(scaled\_mouse\_pos, mouse\_clicked, dt)

        if self.buttons["back"].clicked:

            self.save\_controls()

            self.parent.show\_window("main\_menu")

    def draw(self):

        self.display\_surface.fill((8, 169, 252))

        self.display\_surface.blit(self.title\_image, (195, 5))

        self.display\_surface.blit(self.text, (150, 44))

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        for action\_button in self.action\_buttons.values():

            self.display\_surface.blit(

                action\_button.current\_image, action\_button.rect.topleft

            )

            self.display\_surface.blit(

                action\_button.key\_image,

                (action\_button.rect.right + 10, action\_button.rect.top - 12),

            )

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

    def save\_controls(self):

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            new\_settings\_json = load\_json(settings\_json)

            new\_settings\_json["key\_bindings"] = [

                action\_button.key for action\_button in self.action\_buttons.values()

            ]

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="w"

        ) as settings\_json:

            dump(new\_settings\_json, settings\_json, indent=4)

        self.parent.config.reload()

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerDashState(PlayerAbilityState):

    def enter(self):

        super().enter()

        move\_input = self.player.input.get\_axis\_raw()

        move\_input.scale\_to\_length(self.player.dash\_speed)

from \_\_future\_\_ import annotations

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

class PlayerIdleState(PlayerGroundedState):

    def update(self, dt: float):

        super().update(dt)

        if self.move\_input.x:

            self.player.state\_machine.change\_state(self.player.run\_state)

from \_\_future\_\_ import annotations

from time import perf\_counter

import typing

from nea\_game.calc.lerp import lerp

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

from nea\_game.states.player\_state import PlayerState

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerInAirState(PlayerState):

    move\_input: Vector2D

    jump\_input\_time: float

    def \_\_init\_\_(self, player: Player, state\_name: str):

        super().\_\_init\_\_(player, state\_name)

        self.jump\_input\_time = 0

    def input\_handler(self):

        self.move\_input = self.player.input.get\_axis\_raw()

    def update(self, dt: float):

        if self.player.input.get\_axis\_raw().x:

            if (

                self.player.is\_touching\_wall == self.player.input.get\_axis\_raw().x

                and self.player.rb.velocity.y > 0

            ):

                self.player.state\_machine.change\_state(self.player.slide\_state)

        # Coyote Time

        if (

            self.player.input.get\_action\_down(PlayerActionSpace.UP)

            and perf\_counter() - self.start\_time < self.player.coyote\_time

            and issubclass(

                type(self.player.state\_machine.previous\_state), PlayerGroundedState

            )

        ):

            self.player.state\_machine.change\_state(self.player.jump\_state)

        # Jump Buffering

        if self.player.input.get\_action\_down(PlayerActionSpace.UP):

            self.jump\_input\_time = perf\_counter()

        if self.player.is\_grounded:

            if perf\_counter() - self.jump\_input\_time < self.player.jump\_buffer\_time:

                self.player.state\_machine.change\_state(self.player.jump\_state)

            elif self.player.rb.velocity.x == 0:

                self.player.state\_machine.change\_state(self.player.idle\_state)

            else:

                self.player.state\_machine.change\_state(self.player.run\_state)

        else:

            target\_speed = self.move\_input.x \* self.player.x\_run\_speed

            if self.player.state\_machine.previous\_state == self.player.wall\_jump\_state:

                if (

                    perf\_counter() - self.player.state\_machine.previous\_state.start\_time

                    < self.player.wall\_jump\_time

                ):

                    target\_speed = lerp(

                        self.player.rb.velocity.x,

                        target\_speed,

                        self.player.wall\_jump\_lerp,

                    )

            speed\_difference = target\_speed - self.player.rb.velocity.x

            acceleration\_rate = (

                self.player.acceleration\_rate

                if abs(target\_speed) > 0

                else self.player.deceleration\_rate

            )

            if (

                self.player.state\_machine.previous\_state == self.player.jump\_state

                and abs(self.player.rb.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                acceleration\_rate \*= self.player.jump\_hang\_acceleration\_mult

                target\_speed \*= self.player.jump\_hang\_max\_speed\_mult

            movement = pow(

                abs(speed\_difference) \* acceleration\_rate, self.player.velocity\_power

            ) \* sign(speed\_difference)

            self.player.rb.add\_force(Vector2D(1, 0).scale(movement), dt)

            if abs(target\_speed) == 0:

                friction = sign(self.player.rb.velocity.x) \* min(

                    abs(self.player.rb.velocity.x), self.player.friction

                )

                self.player.rb.add\_force(

                    Vector2D(1, 0).scale(friction), force\_mode=ForceMode.IMPULSE

                )

            if (

                self.player.state\_machine.previous\_state

                in (self.player.jump\_state, self.player.wall\_jump\_state)

                and abs(self.player.rb.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                gravity\_scale = (

                    self.player.rb.gravity\_scale \* self.player.jump\_hang\_gravity\_mult

                )

            elif self.player.rb.velocity.y > 0:

                gravity\_scale = (

                    self.player.rb.gravity\_scale \* self.player.jump\_fast\_fall\_mult

                )

            else:

                gravity\_scale = self.player.rb.gravity\_scale

            self.player.rb.add\_force(

                Vector2D(0, 1).scale(gravity\_scale),

                dt,

                ForceMode.ACCELERATION,

            )

            self.player.rb.velocity = Vector2D(

                self.player.rb.velocity.x,

                min(self.player.rb.velocity.y, self.player.max\_fall),

            )

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerJumpState(PlayerAbilityState):

    def enter(self):

        super().enter()

        self.player.rb.add\_force(

            Vector2D(0, -1).scale(self.player.jump\_force), force\_mode=ForceMode.IMPULSE

        )

        self.is\_ability\_done = True

from \_\_future\_\_ import annotations

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

class PlayerLandState(PlayerGroundedState):

    def update(self, dt: float):

        super().update(dt)

        if self.player.rb.velocity.x == 0 and self.player.input.get\_axis\_raw().x == 0:

            self.player.state\_machine.change\_state(self.player.idle\_state)

        if self.move\_input.x:

            self.player.state\_machine.change\_state(self.player.run\_state)

from \_\_future\_\_ import annotations

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

class PlayerRunState(PlayerGroundedState):

    def update(self, dt: float):

        super().update(dt)

        if self.player.rb.velocity.x == 0 and self.move\_input.x == 0:

            self.player.state\_machine.change\_state(self.player.idle\_state)

        target\_speed = self.move\_input.x \* self.player.x\_run\_speed

        speed\_difference = target\_speed - self.player.rb.velocity.x

        acceleration\_rate = (

            self.player.acceleration\_rate \* self.player.air\_acceleration\_multiplier

            if abs(target\_speed) > 0

            else self.player.deceleration\_rate

        )

        movement = pow(

            abs(speed\_difference) \* acceleration\_rate, self.player.velocity\_power

        ) \* sign(speed\_difference)

        self.player.rb.add\_force(Vector2D(1, 0).scale(movement), dt)

        if abs(target\_speed) == 0:

            friction = sign(self.player.rb.velocity.x) \* min(

                abs(self.player.rb.velocity.x), self.player.friction

            )

            self.player.rb.add\_force(

                Vector2D(self.move\_input.x, 0).scale(friction),

                force\_mode=ForceMode.IMPULSE,

            )

from \_\_future\_\_ import annotations

from nea\_game.calc.vector2d import Vector2D

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.states.player\_state import PlayerState

class PlayerSlideState(PlayerState):

    slide\_direction: int

    move\_input: Vector2D

    def enter(self):

        super().enter()

        self.slide\_direction = self.player.direction

    def input\_handler(self):

        self.move\_input = self.player.input.get\_axis\_raw()

    def update(self, dt: float):

        if self.player.input.get\_action\_down(PlayerActionSpace.UP):

            self.player.state\_machine.change\_state(self.player.wall\_jump\_state)

        if (

            not self.player.is\_touching\_wall

            or self.move\_input.x != self.slide\_direction

        ):

            self.player.state\_machine.change\_state(self.player.in\_air\_state)

        if self.player.is\_grounded:

            self.player.state\_machine.change\_state(self.player.idle\_state)

        if not self.is\_exiting\_state:

            self.player.rb.velocity = Vector2D(self.player.rb.velocity.x, self.player.wall\_slide\_velocity)

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerWallJumpState(PlayerAbilityState):

    def enter(self):

        super().enter()

        force = Vector2D(

            self.player.wall\_jump\_force.x \* self.player.direction \* -1,

            self.player.wall\_jump\_force.y \* -1,

        )

        if sign(self.player.rb.velocity.x) != sign(force.x):

            force = Vector2D(force.x - self.player.rb.velocity.x, force.y)

        if self.player.rb.velocity.y > 0:

            force = Vector2D(force.x, force.y - self.player.rb.velocity.y)

        self.player.rb.add\_force(force, force\_mode=ForceMode.IMPULSE)

        self.is\_ability\_done = True

from time import perf\_counter

from nea\_game.calc.vector2d import Vector2D

from nea\_game.states.player\_state import PlayerState

class PlayerAbilityState(PlayerState):

    move\_input: Vector2D

    start\_time: float

    is\_ability\_done: bool

    def enter(self):

        self.start\_time = perf\_counter()

        self.is\_ability\_done = False

    def update(self, dt: float):

        super().update(dt)

        if self.is\_ability\_done:

            if self.player.is\_grounded and self.player.rb.velocity.y <= 0:

                self.player.state\_machine.change\_state(self.player.idle\_state)

            else:

                self.player.state\_machine.change\_state(self.player.in\_air\_state)

from \_\_future\_\_ import annotations

from nea\_game.calc.vector2d import Vector2D

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.states.player\_state import PlayerState

class PlayerGroundedState(PlayerState):

    move\_input: Vector2D

    def input\_handler(self):

        self.move\_input = self.player.input.get\_axis\_raw()

    def update(self, dt: float):

        super().update(dt)

        if self.player.input.get\_action\_down(PlayerActionSpace.UP):

            self.player.state\_machine.change\_state(self.player.jump\_state)

        if not self.player.is\_grounded:

            self.player.state\_machine.change\_state(self.player.in\_air\_state)

from enum import Enum

class PlayerActionSpace(Enum):

    UP = 0

    DOWN = 1

    LEFT = 2

    RIGHT = 3

    DASH = 4

from nea\_game.states.player\_state import PlayerState

class StateMachine:

    def \_\_init\_\_(self, starting\_state: PlayerState) -> None:

        """A state machine for an entity

        Args:

            starting\_state (State): The starting state for the entity

        """

        self.current\_state = starting\_state

        self.previous\_state = starting\_state

    def change\_state(self, new\_state: PlayerState) -> None:

        """Provides functionlaity for changing state

        Args:

            new\_state (State): The new state that the entity will change to

        """

        self.current\_state.exit()

        self.previous\_state = self.current\_state

        self.current\_state = new\_state

        self.current\_state.enter()

    def get\_current\_state(self) -> PlayerState:

        """A getter for current\_state

        Returns:

            State: The current state of the entity

        """

        return self.current\_state

from \_\_future\_\_ import annotations

import typing

from time import perf\_counter

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerState:

    def \_\_init\_\_(self, player: Player, state\_name: str) -> None:

        """A template PlayerState class

        Args:

            player (Player): The entity that the PlayerState belongs to

            state\_name (str): The string representation of the PlayerState

        """

        self.player = player

        self.state\_name: str = state\_name

        self.start\_time: float

        self.is\_exiting\_state: bool

    def enter(self) -> None:

        """Called when entering the PlayerPlayerState"""

        self.start\_time = perf\_counter()

        self.is\_exiting\_state = False

    def exit(self) -> None:

        """Called when leaving the PlayerPlayerState"""

        self.is\_exiting\_state = True

    def input\_handler(self):

        """Handles the inputs"""

    def update(self, dt: float) -> None:

        """Called each frame"""

{

    "x\_resolution": 1920,

    "y\_resolution": 1080,

    "fps": 60,

    "key\_bindings": [

        119,

        115,

        97,

        100,

        32

    ],

    "unlocked\_levels": [

        true,

        false

    ]

}

"""Provides an interface for storing and loading game wide values"""

from json import load

from pathlib import Path

from time import strftime

class NeaGameConfig:

    """Handles loading of the config.json file"""

    debug: bool

    debug\_file: str

    directories: dict[str, Path]

    resoloution: tuple[int, int]

    internal\_resoloution: tuple[int, int]

    fps: int

    internal\_fps: int

    chunk\_size: int

    key\_bindings: list[int]

    unlocked\_levels: list[bool]

    def \_\_init\_\_(self):

        # Debug

        self.debug = False

        debug\_filename = "Test"

        self.debug\_file = (strftime("%m-%d-%Y")) + "-" + debug\_filename + ".prof"

        # Directories

        platformer\_folder = Path(\_\_file\_\_).absolute().parent

        game\_folder = platformer\_folder.parent

        assets\_folder = game\_folder / "assets"

        background\_folder = assets\_folder / "background"

        buttons\_folder = assets\_folder / "buttons"

        player\_folder = assets\_folder / "player"

        worlds\_folder = assets\_folder / "worlds"

        self.directories = {

            "game": game\_folder,

            "assets": assets\_folder,

            "background": background\_folder,

            "buttons": buttons\_folder,

            "player": player\_folder,

            "worlds": worlds\_folder,

            "platformer": platformer\_folder,

        }

        # Display

        width = self.get\_int\_setting("x\_resolution")

        height = self.get\_int\_setting("y\_resolution")

        self.resoloution = (width, height)

        ds\_width = 384

        ds\_height = 216

        self.internal\_resoloution = (ds\_width, ds\_height)

        self.fps = self.get\_int\_setting("fps")

        self.internal\_fps = 60

        self.key\_bindings = self.get\_integer\_list\_setting("key\_bindings")

        self.unlocked\_levels = self.get\_bool\_list\_setting("unlocked\_levels")

    def get\_int\_setting(self, setting: str) -> int:

        """\_summary\_

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [int]

        Returns:

            int: The value of the specified setting

        """

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if isinstance(settings[setting], int):

                return settings[setting]

            raise ValueError

    def get\_float\_setting(self, setting: str) -> float:

        """\_summary\_

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [float]  or [int]

        Returns:

            list[int]: The value of the specified setting

        """

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if isinstance(settings[setting], (float, int)):

                return settings[setting]

            raise ValueError

    def get\_integer\_list\_setting(self, setting: str) -> list[int]:

        """Gets the relevant setting from config.json that is a list of integers

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [float] or [int]

        Returns:

            list[int]: The value of the specified setting

        """

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if all(isinstance(element, int) for element in settings[setting]):

                return settings[setting]

            raise ValueError

    def get\_bool\_list\_setting(self, setting: str) -> list[bool]:

        """Gets the relevant setting from config.json that is a list of bools

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [bool]

        Returns:

            list[bool]: The value of the specified setting

        """

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if all(isinstance(element, bool) for element in settings[setting]):

                return settings[setting]

            raise ValueError

    def reload(self):

        """Reloads the settings from config.json"""

        width = self.get\_int\_setting("x\_resolution")

        height = self.get\_int\_setting("y\_resolution")

        self.resoloution = (width, height)

        self.fps = self.get\_int\_setting("fps")

        self.key\_bindings = self.get\_integer\_list\_setting("key\_bindings")

from sys import exit as sys\_exit

from pygame.event import Event

import pygame

from nea\_game.config import NeaGameConfig

from nea\_game.gui.root import Root

from nea\_game.menu.main\_menu import MainMenu

class NeaGame(Root):

    def \_\_init\_\_(self, config: NeaGameConfig):

        self.config = config

        pygame.display.set\_caption("NEA Game")

        # pygame.display.set\_icon()

        super().\_\_init\_\_(

            self.config.resoloution,

            self.config.internal\_resoloution,

            self.config.fps,

        )

        self.windows["main\_menu"] = MainMenu(

            self,

            self.screen,

            self.display\_surface,

            config.directories["assets"] / "title.png",

            config.directories["background"] / "sky\_mountain",

            config.directories["buttons"] / "main\_menu",

        )

        self.show\_window("main\_menu")

    def update(self):

        self.get\_events()

        super().update()

    def get\_events(self):

        events: list[Event]

        events = []

        for event in pygame.event.get():

            if event.type == pygame.QUIT or (

                event.type == pygame.KEYDOWN and event.key == pygame.K\_ESCAPE

            ):

                pygame.quit()

                sys\_exit()

            else:

                events.append(event)

        self.active\_window.event\_handler(events)

from nea\_game.nea\_game import NeaGame

from nea\_game.config import NeaGameConfig

def app():

    nea\_game = NeaGame(config)

    running = True

    while running:

        nea\_game.update()

if \_\_name\_\_ == "\_\_main\_\_":

    config = NeaGameConfig()

    if config.debug:

        import ez\_profile

    app()