```python
from pathlib import Path
from data.utils.load_helpers import *

module_path = Path(__file__).parent
GRAPHICS = load_all_gfx((module_path / '../resources/graphics').resolve())
FONTS = load_all_fonts((module_path / '../resources/fonts').resolve())
SFX = load_all_sfx((module_path / '../resources/sfx').resolve())
MUSIC = load_all_music((module_path / '../resources/music').resolve())

DEFAULT_FONT=FONTS['vhs-gothic']
DEFAULT_FONT.strong = True
DEFAULT_FONT.strength = 0.05
```

```python
import pygame
from enum import IntEnum, StrEnum, auto

BG_COLOUR = (0, 0, 0)
PAUSE_COLOUR = (50, 50, 50, 128)
OVERLAY_COLOUR_LIGHT = (*pygame.Color('0xf14e52').rgb, 128)
OVERLAY_COLOUR_DARK = (*pygame.Color('0x9b222b').rgb, 192)
SCREEN_SIZE = (1200, 600)
# SCREEN_SIZE = (600, 600)
SCREEN_FLAGS = pygame.HWSURFACE | pygame.DOUBLEBUF | pygame.RESIZABLE | pygame.OPENGL
STARTING_SQUARE_SIZE = (SCREEN_SIZE[1] * 0.64) / 8 #Board height divded by 8
EMPTY_BB = 0
A_FILE_MASK = 0b11111111101111111110111111111011111111101111111110111111111011111111101111111110
J_FILE_MASK = 0b01111111110111111111011111111101111111110111111111011111111101111111110111111111
ONE_RANK_MASK = 0b11111111111111111111111111111111111111111111111111111111111111111111110000000000
EIGHT_RANK_MASK = 0b00000000001111111111111111111111111111111111111111111111111111111111111111111111
TEST_MASK = 0b00000000100000000010000000000000000000000000000000000000000000000010000000001000000000

GAMES_PER_PAGE = 10

class CursorMode(IntEnum):
    ARROW = auto()
    IBEAM = auto()
    OPENHAND = auto()
    CLOSEDHAND = auto()
    NO = auto()

class ShaderType(StrEnum):
    BASE = auto()
    SHAKE = auto()
    BLOOM = auto()
    GRAYSCALE = auto()
    CRT = auto()
    RAYS = auto()
    CHROMATIC_ABBREVIATION = auto()
    BACKGROUND_WAVES = auto()
    BACKGROUND_BALATRO = auto()
    BACKGROUND_LASERS = auto()
    BACKGROUND_GRADIENT = auto()
    BACKGROUND_NONE = auto()

    _BLUR = auto()
    _HIGHLIGHT_BRIGHTNESS = auto()
    _HIGHLIGHT_COLOUR = auto()
    _CALIBRATE = auto()
    _LIGHTMAP = auto()
    _SHADOWMAP = auto()
    _OCCLUSION = auto()
    _BLEND = auto()
    _CROP = auto()

SHADER_MAP = {
    'default': [
        ShaderType.BLOOM
    ],
    'retro': [
        ShaderType.CRT
    ],
    'really_retro': [
        ShaderType.CRT,
        ShaderType.GRAYSCALE
    ],
}

class TranspositionFlag(StrEnum):
    LOWER = auto()
    EXACT = auto()
    UPPER = auto()

class Miscellaneous(StrEnum):
    PLACEHOLDER = auto()
    DRAW = auto()

class WidgetState(StrEnum):
    BASE = auto()
    HOVER = auto()
    PRESS = auto()

BLUE_BUTTON_COLOURS = {
    WidgetState.BASE: ['0x1c2638', '0x23495d', '0x39707a', '0x95e0cc'],
    WidgetState.HOVER: ['0xdaf2e9', '0x23495d', '0x39707a', '0x95e0cc'],
    WidgetState.PRESS: ['0xdaf2e9', '0x1c2638', '0x23495d', '0x39707a']
}

INPUT_COLOURS = {
    WidgetState.BASE: ['0x1c2638', '0x39707a', '0x23495d', '0x95e0cc'],
    WidgetState.HOVER: ['0xdaf2e9', '0x39707a', '0x23495d', '0x95e0cc'],
    WidgetState.PRESS: ['0xdaf2e9', '0x23495d', '0x1c2638', '0x39707a']
}

RED_BUTTON_COLOURS = {
    WidgetState.BASE: ['0x000000', '0x1c2638', '0x9b222b', '0xf14e52'],
    WidgetState.HOVER: ['0xdaf2e9', '0x1c2638', '0x9b222b', '0xf14e52'],
    WidgetState.PRESS: ['0xdaf2e9', '0x23495d', '0xf14e52', '0x95e0cc']
}

LOCKED_RED_BUTTON_COLOURS = {
    WidgetState.BASE: ['0x000000', '0x000000', '0x1c2638', '0x23495d'],
    WidgetState.HOVER: ['0xdaf2e9', '0x000000', '0x1c2638', '0x23495d'],
    WidgetState.PRESS: ['0xdaf2e9', '0x1c2638', '0x23495d', '0xf14e52']
}
```

```

LOCKED_BLUE_BUTTON_COLOURS = {
    WidgetState.BASE: ['0x000000', '0x000000', '0x1c2638', '0x23495d'],
    WidgetState.HOVER: ['0xdaf2e9', '0x000000', '0x1c2638', '0x23495d'],
    WidgetState.PRESS: ['0xdaf2e9', '0x1c2638', '0x23495d', '0x39707a']
}

class StatusText(StrEnum):
    PLAYER_MOVE = auto()
    CPU_MOVE = auto()
    WIN = auto()
    DRAW = auto()

class EditorEventType(StrEnum):
    MENU_CLICK = auto()
    PICK_PIECE_CLICK = auto()
    ROTATE_PIECE_CLICK = auto()
    COPY_CLICK = auto()
    EMPTY_CLICK = auto()
    RESET_CLICK = auto()
    BLUE_START_CLICK = auto()
    RED_START_CLICK = auto()
    START_CLICK = auto()
    CONFIG_CLICK = auto()
    ERASE_CLICK = auto()
    MOVE_CLICK = auto()
    HELP_CLICK = auto()

class ReviewEventType(StrEnum):
    MENU_CLICK = auto()
    PREVIOUS_CLICK = auto()
    NEXT_CLICK = auto()
    HELP_CLICK = auto()

class BrowserEventType(StrEnum):
    MENU_CLICK = auto()
    BROWSER_STRIP_CLICK = auto()
    COPY_CLICK = auto()
    DELETE_CLICK = auto()
    REVIEW_CLICK = auto()
    FILTER_COLUMN_CLICK = auto()
    FILTER_ASCEND_CLICK = auto()
    PAGE_CLICK = auto()
    HELP_CLICK = auto()

class GameEventType(StrEnum):
    BOARD_CLICK = auto()
    PIECE_CLICK = auto()
    PAUSE_CLICK = auto()
    MENU_CLICK = auto()
    GAME_CLICK = auto()
    HELP_CLICK = auto()
    TUTORIAL_CLICK = auto()
    RESIGN_CLICK = auto()
    DRAW_CLICK = auto()
    REVIEW_CLICK = auto()
    PIECE_DROP = auto()
    UPDATE_PIECES = auto()
    ROTATE_PIECE = auto()
    SET_LASER = auto()
    TIMER_END = auto()

```

```python
class MenuEventType(StrEnum):
    CONFIG_CLICK = auto()
    SETTINGS_CLICK = auto()
    BROWSER_CLICK = auto()
    QUIT_CLICK = auto()
    CREDITS_CLICK = auto()

class SettingsEventType(StrEnum):
    RESET_DEFAULT = auto()
    RESET_USER = auto()
    MENU_CLICK = auto()
    COLOUR_SLIDER_SLIDE = auto()
    COLOUR_SLIDER_CLICK = auto()
    COLOUR_PICKER_HOVER = auto()
    PRIMARY_COLOUR_PICKER_CLICK = auto()
    SECONDARY_COLOUR_PICKER_CLICK = auto()
    PRIMARY_COLOUR_BUTTON_CLICK = auto()
    SECONDARY_COLOUR_BUTTON_CLICK = auto()
    VOLUME_SLIDER_SLIDE = auto()
    VOLUME_SLIDER_CLICK = auto()
    SHADER_PICKER_CLICK = auto()
    OPENGL_CLICK = auto()
    DROPDOWN_CLICK = auto()
    PARTICLES_CLICK = auto()

class ConfigEventType(StrEnum):
    GAME_CLICK = auto()
    MENU_CLICK = auto()
    FEN_STRING_TYPE = auto()
    TIME_TYPE = auto()
    TIME_CLICK = auto()
    PVP_CLICK = auto()
    PVC_CLICK = auto()
    CPU_DEPTH_CLICK = auto()
    PRESET_CLICK = auto()
    SETUP_CLICK = auto()
    COLOUR_CLICK = auto()
    HELP_CLICK = auto()

class Colour(IntEnum):
    BLUE = 0
    RED = 1

    def get_flipped_colour(self):
        if self == Colour.BLUE:
            return Colour.RED
        elif self == Colour.RED:
            return Colour.BLUE

class Piece(StrEnum):
    SPHINX = 's'
    PYRAMID = 'p'
    ANUBIS = 'n'
    SCARAB = 'r'
    PHAROAH = 'f'

class Score(IntEnum):
    PHAROAH = 0
    SPHINX = 0
    PYRAMID = 100
    ANUBIS = 110
    SCARAB = 200

    MOVE = 4
    POSITION = 11
    PHAROAH_SAFETY = 31
    CHECKMATE = 100000
    INFINITE = 6969696969

class Rank(IntEnum):
    ONE = 0
    TWO = 1
    THREE = 2
    FOUR = 3
    FIVE = 4
    SIX = 5
    SEVEN = 6
    EIGHT = 7

class File(IntEnum):
    A = 0
    B = 1
    C = 2
    D = 3
    E = 4
    F = 5
    G = 6
    H = 7
    I = 8
    J = 9

class Rotation(StrEnum):
    UP = 'a'
    RIGHT = 'b'
    DOWN = 'c'
    LEFT = 'd'

    def to_angle(self):
        if self == Rotation.UP:
            return 0
        elif self == Rotation.RIGHT:
            return 270
        elif self == Rotation.DOWN:
            return 180
        elif self == Rotation.LEFT:
            return 90

    def get_clockwise(self):
        if self == Rotation.UP:
            return Rotation.RIGHT
        elif self == Rotation.RIGHT:
            return Rotation.DOWN
        elif self == Rotation.DOWN:
            return Rotation.LEFT
        elif self == Rotation.LEFT:
            return Rotation.UP

    def get_anticlockwise(self):
        if self == Rotation.UP:
            return Rotation.LEFT
        elif self == Rotation.RIGHT:
            return Rotation.UP
        elif self == Rotation.DOWN:
            return Rotation.RIGHT
        elif self == Rotation.LEFT:
            return Rotation.DOWN

    def get_opposite(self):
        return self.get_clockwise().get_clockwise()

class RotationIndex(IntEnum):
    FIRSTBIT = 0
    SECONDBIT = 1

class RotationDirection(StrEnum):
    CLOCKWISE = 'cw'
    ANTICLOCKWISE = 'acw'

    def get_opposite(self):
        if self == RotationDirection.CLOCKWISE:
            return RotationDirection.ANTICLOCKWISE
        elif self == RotationDirection.ANTICLOCKWISE:
            return RotationDirection.CLOCKWISE

class MoveType(StrEnum):
    MOVE = 'm'
    ROTATE = 'r'

class LaserType(IntEnum):
    END = 0
    STRAIGHT = 1
    CORNER = 2

class LaserDirection(IntEnum):
    FROM_TOP = 1
    FROM_RIGHT = 2
    FROM_BOTTOM = 3
    FROM_LEFT = 4
```

```python
import pygame
from data.components.widget_group import WidgetGroup
from data.managers.logs import initialise_logger
from data.managers.cursor import CursorManager
from data.managers.animation import animation
from data.managers.window import window
from data.managers.audio import audio
from data.managers.theme import theme
from data.assets import DEFAULT_FONT

logger = initialise_logger(__file__)

FPS = 60
SHOW_FPS = False
start_ticks = pygame.time.get_ticks()

class Control:
    def __init__(self):
        self.done = False
        self._clock = pygame.time.Clock()

    def setup_states(self, state_dict, start_state):
        self.state_dict = state_dict
        self.state_name = start_state

        self.state = self.state_dict[self.state_name]
        self.state.startup()

    def flip_state(self):
        self.state.done = False
        persist = self.state.cleanup()

        previous, self.state_name = self.state_name, self.state.next

        self.state = self.state_dict[self.state_name]
        self.state.previous = previous
        self.state.startup(persist)

    def update(self):
        if self.state.quit:
            self.done = True
        elif self.state.done:
            self.flip_state()

        self._clock.tick(FPS)
        animation.set_delta_time()

        self.state.update()

        if SHOW_FPS:
            self.draw_fps()

        window.update()

    def main_game_loop(self):
        while not self.done:
            self.event_loop()
            self.update()

    def update_window(self, resize=False):
        if resize:
            self.update_native_window_size()
            window.handle_resize()
            self.state.handle_resize()

        self.update()

    def draw_fps(self):
        fps = str(int(self._clock.get_fps()))
        DEFAULT_FONT.strength = 0.1
        DEFAULT_FONT.render_to(window.screen, (0, 0), fps, fgcolor=theme['textError'], size=15)

    def update_native_window_size(self):
        x, y = window.size

        max_window_x = 100000
        max_window_y = x / 1.4
        min_window_x = 400
        min_window_y = min_window_x/1.4

        if x / y < 1.4:
            min_window_x = x

        min_window_size = (min_window_x, min_window_y)
        max_window_size = (max_window_x, max_window_y)
        window.minimum_size = min_window_size
        window.maximum_size = max_window_size

    def event_loop(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.done = True

            if event.type == pygame.MOUSEBUTTONDOWN and event.button != 1: # ONLY PROCESS LEFT CLICKS
                return

            self.state.get_event(event)

class _State:
    def __init__(self):
        self.next = None
        self.previous = None
        self.done = False
        self.quit = False
        self.persist = {}

        self._cursor = CursorManager()
        self._widget_group = None

    def startup(self, widgets=None, music=None):
        if widgets:
            self._widget_group = WidgetGroup(widgets)
            self._widget_group.handle_resize(window.size)

        if music:
            audio.play_music(music)

        logger.info(f'starting {self.__class__.__name__.lower()}.py')

    def cleanup(self):
        logger.info(f'cleaning {self.__class__.__name__.lower()}.py')

    def draw(self):
        raise NotImplementedError

    def get_event(self, event):
        raise NotImplementedError

    def handle_resize(self):
        self._widget_group.handle_resize(window.size)

    def update(self, **kwargs):
        self.draw()
```

```
import pygame
import threading
import sys
from pathlib import Path
from data.utils.load_helpers import load_gfx, load_sfx
from data.managers.window import window
from data.managers.audio import audio

FPS = 30
start_ticks = pygame.time.get_ticks()
logo_gfx_path = (Path(__file__).parent / '../resources/graphics/gui/icons/logo/logo.png').resolve()
sfx_path_1 = (Path(__file__).parent / '../resources/sfx/loading_screen/loading_screen_1.wav').resolve()
sfx_path_2 = (Path(__file__).parent / '../resources/sfx/loading_screen/
    loading_screen_2.wav').resolve()

def easeOutBack(progress):
    """
    Represents a cubic function for easing the logo position.
    Starts quickly and has small overshoot, then ends slowly.

    Args:
        progress (float): x-value for cubic function ranging from 0-1.

    Returns:
        float: 2.70x^3 + 1.70x^2 + 0x + 1, where x is time elapsed.
    """
    c2 = 1.70158
    c3 = 2.70158

    return c3 * ((progress - 1) ** 3) + c2 * ((progress - 1) ** 2) + 1

class LoadingScreen:
    def __init__(self, target_func):
        """
        Creates new thread, and sets the load_state() function as its target.
        Then starts draw loop for the loading screen.

        Args:
            target_func (Callable): function to be run on thread.
        """
        self._clock = pygame.time.Clock()
        self._thread = threading.Thread(target=target_func)
        self._thread.start()

        self._logo_surface = load_gfx(logo_gfx_path)
        self._logo_surface = pygame.transform.scale(self._logo_surface, (96, 96))
        audio.play_sfx(load_sfx(sfx_path_1))
        audio.play_sfx(load_sfx(sfx_path_2))

        self.run()

    @property
    def logo_position(self):
        duration = 1000
        displacement = 50
        elapsed_ticks = pygame.time.get_ticks() - start_ticks
        progress = min(1, elapsed_ticks / duration)
        center_pos = ((window.screen.size[0] - self._logo_surface.size[0]) / 2, (
    window.screen.size[1] - self._logo_surface.size[1]) / 2)

        return (center_pos[0], center_pos[1] + displacement - displacement *
    easeOutBack(progress))

    @property
    def logo_opacity(self):
        return min(255, (pygame.time.get_ticks() - start_ticks) / 5)

    @property
    def duration_not_over(self):
        return (pygame.time.get_ticks() - start_ticks) < 1500

    def event_loop(self):
        """
        Handles events for the loading screen, no user input is taken except to
    quit the game.
```

```
        """
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

    def draw(self):
        """
        Draws logo to screen.
        """
        window.screen.fill((0, 0, 0))

        self._logo_surface.set_alpha(self.logo_opacity)
        window.screen.blit(self._logo_surface, self.logo_position)

        window.update()

    def run(self):
        """
        Runs while the thread is still setting up our screens, or the minimum loading screen duration is not reached yet.
        """
        while self._thread.is_alive() or self.duration_not_over:
            self.event_loop()
            self.draw()
            self._clock.tick(FPS)
```

```
from sys import platform
# Initialises Pygame
import data.setup

# Windows OS requires some configuration for Pygame to scale GUI continuously while window is being resized
if platform == 'win32':
    import data.windows_setup as win_setup

from data.loading_screen import LoadingScreen

states = [None, None]

def load_states():
    """
    Initialises instances of all screens, executed on another thread with results being stored to the main thread by modifying a mutable such as the states list
    """
    from data.control import Control
    from data.states.game.game import Game
    from data.states.menu.menu import Menu
    from data.states.settings.settings import Settings
    from data.states.config.config import Config
    from data.states.browser.browser import Browser
    from data.states.review.review import Review
    from data.states.editor.editor import Editor

    state_dict = {
        'menu': Menu(),
        'game': Game(),
        'settings': Settings(),
        'config': Config(),
        'browser': Browser(),
        'review': Review(),
        'editor': Editor()
```

```python
    }

    app = Control()

    states[0] = app
    states[1] = state_dict

loading_screen = LoadingScreen(load_states)

def main():
    """
    Executed by run.py, starts main game loop
    """
    app, state_dict = states

    if platform == 'win32':
        win_setup.set_win_resize_func(app.update_window)

    app.setup_states(state_dict, 'menu')
    app.main_game_loop()
```

```python
import pygame

pygame.mixer.init()
pygame.init()

pygame.display.gl_set_attribute(pygame.GL_CONTEXT_MAJOR_VERSION, 3)
pygame.display.gl_set_attribute(pygame.GL_CONTEXT_MINOR_VERSION, 3)
pygame.display.gl_set_attribute(pygame.GL_CONTEXT_PROFILE_MASK, pygame.
    GL_CONTEXT_PROFILE_CORE)
pygame.display.gl_set_attribute(pygame.GL_CONTEXT_FORWARD_COMPATIBLE_FLAG, True)
```

```python
import win32gui
import win32con
import os
import ctypes
import sys

def wndProc(oldWndProc, draw_callback, hWnd, message, wParam, lParam):
    if message == win32con.WM_SIZING or message == win32con.WM_TIMER: # Don't know
     what WM_TIMER does
        draw_callback(resize=True)
        win32gui.RedrawWindow(hWnd, None, None, win32con.RDW_INVALIDATE | win32con
    .RDW_ERASE)
    elif message == win32con.WM_MOVE:
        draw_callback(resize=False)

    return win32gui.CallWindowProc(oldWndProc, hWnd, message, wParam, lParam)

def set_win_resize_func(resize_function):
    oldWndProc = win32gui.SetWindowLong(win32gui.GetForegroundWindow(), win32con.
    GWL_WNDPROC, lambda *args: wndProc(oldWndProc, resize_function, *args))

user32 = ctypes.windll.user32
user32.SetProcessDPIAware() # To deal with Windows High Text Size / Low Display
    Resolution Settings

if os.name != 'nt' or sys.getwindowsversion()[0] < 6:
    raise NotImplementedError("Incompatible OS!")
```

```
{
    "primaryBoardColour": "0xB98766",
    "secondaryBoardColour": "0xF3D8B8",
    "laserColourBlue": "0x0000ff",
    "laserColourRed": "0xff0000",
    "displayMode": "windowed",
    "musicVolume": 0.5,
    "sfxVolume": 0.5,
    "particles": true,
    "opengl": true,
    "shader": "default"
}
```

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
      "simple": {
        "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        "datefmt": "%Y-%m-%d %H:%M:%S"
      }
    },

    "handlers": {
      "console": {
        "class": "logging.StreamHandler",
        "formatter": "simple",
        "stream": "ext://sys.stdout"
      }
    },

    "root": {
      "level": "INFO",
      "handlers": ["console"],
      "propagate": false
    }
  }
```

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
      "simple": {
        "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
      }
    },

    "handlers": {
      "console": {
        "class": "logging.StreamHandler",
        "level": "DEBUG",
        "formatter": "simple",
        "stream": "ext://sys.stdout"
      },

      "info_file_handler": {
        "class": "logging.handlers.RotatingFileHandler",
        "level": "INFO",
        "formatter": "simple",
        "filename": "info.log",
        "maxBytes": 10485760,
        "backupCount": 20,
        "encoding": "utf8"
      },

      "error_file_handler": {
        "class": "logging.handlers.RotatingFileHandler",
        "level": "ERROR",
        "formatter": "simple",
        "filename": "errors.log",
        "maxBytes": 10485760,
        "backupCount": 20,
        "encoding": "utf8"
      }
    },

    "loggers": {
      "my_module": {
        "level": "ERROR",
        "handlers": ["console"],
        "propagate": false
      }
    },

    "root": {
      "level": "INFO",
      "handlers": ["console", "info_file_handler", "error_file_handler"]
    }
  }
```

```
{
    "colours": {
        "text": {
            "primary": "0xdaf2e9",
            "secondary": "0xf14e52",
            "error": "0xf14e52"
        },
        "fill": {
            "primary": "0x1c2638",
            "secondary": "0xf14e52",
            "tertiary": "0xdaf2e9",
            "error": "0x9b222b"
        },
        "border": {
            "primary": "0x9b222b",
            "secondary": ""
        }
    },
    "dimensions": {
        "borderRadius": 3,
        "borderWidth": 5,
        "margin": 10
    }
}
```

```
{
    "primaryBoardColour": "0xB98766",
    "secondaryBoardColour": "0xF3D8B8",
    "laserColourBlue": "0x0000ff",
    "laserColourRed": "0xff0000",
    "displayMode": "windowed",
    "musicVolume": 0.085,
    "sfxVolume": 0.336,
    "particles": true,
    "opengl": true,
    "shader": "default"
}
```

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

class CircularLinkedList:
    def __init__(self, list_to_convert=None):
        """
        Initialises a CircularLinkedList object.

        Args:
            list_to_convert (list, optional): Creates a linked list from existing items. Defaults to None.
        """
        self._head = None

        if list_to_convert:
            for item in list_to_convert:
                self.insert_at_end(item)

    def __str__(self):
        """
        Returns a string representation of the circular linked list.

        Returns:
            str: Linked list formatted as string.
        """
        if self._head is None:
            return '| empty |'

        characters = '| -> '
        current_node = self._head
        while True:
            characters += str(current_node.data) + ' -> '
            current_node = current_node.next

            if current_node == self._head:
                characters += '|'
                return characters

    def insert_at_beginning(self, data):
        """
        Inserts a node at the beginning of the circular linked list.

        Args:
            data: The data to insert.
        """
        new_node = Node(data)

        if self._head is None:
            self._head = new_node
            new_node.next = self._head
            new_node.previous = self._head
        else:
            new_node.next = self._head
            new_node.previous = self._head.previous
            self._head.previous.next = new_node
            self._head.previous = new_node

            self._head = new_node

    def insert_at_end(self, data):
        """
        Inserts a node at the end of the circular linked list.

        Args:
            data: The data to insert.
        """
        new_node = Node(data)

        if self._head is None:
            self._head = new_node
            new_node.next = self._head
            new_node.previous = self._head
        else:
            new_node.next = self._head
            new_node.previous = self._head.previous
            self._head.previous.next = new_node
            self._head.previous = new_node

    def insert_at_index(self, data, index):
        """
        Inserts a node at a specific index in the circular linked list.
        The head node is taken as index 0.

        Args:
            data: The data to insert.
            index (int): The index to insert the data at.

        Raises:
            ValueError: Index is out of range.
        """
        if index < 0:
            raise ValueError('Invalid index! (CircularLinkedList.insert_at_index)')

        if index == 0 or self._head is None:
            self.insert_at_beginning(data)
        else:
            new_node = Node(data)
            current_node = self._head
            count = 0

            while count < index - 1 and current_node.next != self._head:
                current_node = current_node.next
                count += 1

            if count == (index - 1):
                new_node.next = current_node.next
                new_node.previous = current_node
                current_node.next = new_node
            else:
                raise ValueError('Index out of range! (CircularLinkedList.insert_at_index)')

    def delete(self, data):
        """
        Deletes a node with the specified data from the circular linked list.

        Args:
            data: The data to delete.

        Raises:
            ValueError: No nodes in the list contain the specified data.
        """
        if self._head is None:
            return

        current_node = self._head

        while current_node.data != data:
            current_node = current_node.next

            if current_node == self._head:
                raise ValueError('Data not found in circular linked list! (CircularLinkedList.delete)')

        if self._head.next == self._head:
            self._head = None
        else:
            current_node.previous.next = current_node.next
            current_node.next.previous = current_node.previous

    def data_in_list(self, data):
        """
        Checks if the specified data is in the circular linked list.

        Args:
            data: The data to check.

        Returns:
            bool: True if the data is in the list, False otherwise.
        """
        if self._head is None:
            return False

        current_node = self._head
        while True:
            if current_node.data == data:
                return True

            current_node = current_node.next
            if current_node == self._head:
                return False

    def shift_head(self):
        """
        Shifts the head of the circular linked list to the next node.
        """
        self._head = self._head.next

    def unshift_head(self):
        """
        Shifts the head of the circular linked list to the previous node.
        """
        self._head = self._head.previous

    def get_head(self):
        """
        Gets the head node of the circular linked list.

```

```python
        Returns:
            Node: The head node.
        """
        return self._head
```

```python
import pygame

class Cursor(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((1, 1))
        self.image.fill((255, 0, 0))
        self.rect = self.image.get_rect()

    # def update(self):
    #     self.rect.center = pygame.mouse.get_pos()

    def get_sprite_collision(self, mouse_pos, square_group):
        self.rect.center = mouse_pos
        sprite = pygame.sprite.spritecollideany(self, square_group)

        return sprite
```

```python
from data.constants import GameEventType, SettingsEventType, ConfigEventType, BrowserEventType, EditorEventType

required_args = {
    GameEventType.BOARD_CLICK: ['coords'],
    GameEventType.ROTATE_PIECE: ['rotation_direction'],
    GameEventType.SET_LASER: ['laser_result'],
    GameEventType.UPDATE_PIECES: ['move_notation'],
    GameEventType.TIMER_END: ['active_colour'],
    GameEventType.PIECE_DROP: ['coords', 'piece', 'colour', 'rotation', 'remove_overlay'],
    SettingsEventType.COLOUR_SLIDER_SLIDE: ['colour'],
    SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK: ['colour'],
    SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK: ['colour'],
    SettingsEventType.DROPDOWN_CLICK: ['selected_word'],
    SettingsEventType.VOLUME_SLIDER_CLICK: ['volume', 'volume_type'],
    SettingsEventType.SHADER_PICKER_CLICK: ['data'],
    SettingsEventType.PARTICLES_CLICK: ['toggled'],
    SettingsEventType.OPENGL_CLICK: ['toggled'],
    ConfigEventType.TIME_TYPE: ['time'],
    ConfigEventType.FEN_STRING_TYPE: ['time'],
    ConfigEventType.CPU_DEPTH_CLICK: ['data'],
    ConfigEventType.PVC_CLICK: ['data'],
    ConfigEventType.PRESET_CLICK: ['fen_string'],
    BrowserEventType.BROWSER_STRIP_CLICK: ['selected_index'],
    BrowserEventType.PAGE_CLICK: ['data'],
    EditorEventType.PICK_PIECE_CLICK: ['piece', 'active_colour'],
    EditorEventType.ROTATE_PIECE_CLICK: ['rotation_direction'],
}

class CustomEvent():
    def __init__(self, type, **kwargs):
        self.__dict__.update(kwargs)
        self.type = type

    @classmethod
    def create_event(event_cls, event_type, **kwargs):
        """
        @classmethod Factory method used to instance CustomEvent object, to check for required keyword arguments

        Args:
            event_cls (CustomEvent): Reference to own class.
            event_type: The state EventType.

        Raises:
            ValueError: If required keyword argument for passed event type not present.
            ValueError: If keyword argument passed is not required for passed event type.

        Returns:
            CustomEvent: Initialised CustomEvent instance.
        """
        if event_type in required_args:

            for required_arg in required_args[event_type]:
                if required_arg not in kwargs:
                    raise ValueError(f"Argument '{required_arg}' required for {event_type.name} event (GameEvent.create_event)")

            for kwarg in kwargs:
                if kwarg not in required_args[event_type]:
                    raise ValueError(f"Argument '{kwarg}' not included in required_args dictionary for event '{event_type}'! (GameEvent.create_event)")

            return event_cls(event_type, **kwargs)

        else:
            return event_cls(event_type)
```

```
from data.constants import Colour
from data.states.game.components.move import Move

class GameEntry:
    def __init__(self, game_states, final_fen_string):
        self._game_states = game_states
        self._final_fen_string = final_fen_string

    def __str__(self):
        return f'''
<GameEntry>:
    CPU_ENABLED: {self._game_states['CPU_ENABLED']}
    CPU_DEPTH: {self._game_states['CPU_DEPTH']},
    WINNER: {self._game_states['WINNER']},
    TIME_ENABLED: {self._game_states['TIME_ENABLED']},
    TIME: {self._game_states['TIME']},
    NUMBER_OF_PLY: {len(self._game_states['MOVES'])},
    MOVES: {self.convert_moves(self._game_states['MOVES'])}
    FINAL FEN_STRING: {self._final_fen_string}
    START FEN STRING: {self._game_states['START_FEN_STRING']}
</GameEntry>
        '''

    def convert_to_row(self):
        return (self._game_states['CPU_ENABLED'], self._game_states['CPU_DEPTH'], self._game_states['WINNER'], self._game_states['TIME_ENABLED'], self._game_states['TIME'], len(self._game_states['MOVES']), self.convert_moves(self._game_states['MOVES']), self._game_states['START_FEN_STRING'], self._final_fen_string)

    def convert_moves(self, moves):
        return '|'.join([
            f'{round(move['time'][Colour.BLUE], 4)};{round(move['time'][Colour.RED], 4)};{move['move']}'
            for move in moves
        ])

    @staticmethod
    def parse_moves(move_str):
        moves = move_str.split('|')
        return [
            {
                'blue_time': move.split(';')[0],
                'red_time': move.split(';')[1],
                'move': Move.instance_from_notation(move.split(';')[2]),
                'unparsed_move': move.split(';')[2],
            } for move in moves if move != ''
        ]

# self.states = {
#     'CPU_ENABLED': game_config['CPU_ENABLED'],
#     'CPU_DEPTH': game_config['CPU_DEPTH'],
#     'AWAITING_CPU': False,
#     'WINNER': None,
#     'PAUSED': False,
#     'ACTIVE_COLOUR': Colour.BLUE,
#     'TIME_ENABLED': game_config['TIME_ENABLED'],
#     'TIME': game_config['TIME'],
#     'MOVES': []
# }


#     move_item = {
#     'time': {
#         Colour.BLUE: GAME_WIDGETS['blue_timer'].get_time(),
#         Colour.RED: GAME_WIDGETS['red_timer'].get_time()
#     },
#     'move': move_notation,
#     'laserResult': laser_result
# }
```

```
import pygame
from data.managers.window import window

class WidgetGroup(pygame.sprite.Group):
    def __init__(self, widget_dict):
        super().__init__()

        for value in widget_dict.values():
            if isinstance(value, list):
                for widget in value:
                    self.add(widget)
            elif isinstance(value, dict):
                for widget in value.values():
                    self.add(widget)
            else:
                self.add(value)

    def handle_resize(self, new_surface_size):
        for sprite in self.sprites():
            sprite.set_surface_size(new_surface_size)
```

```python
            sprite.set_image()
            sprite.set_geometry()

    def process_event(self, event):
        for sprite in self.sprites():
            widget_event = sprite.process_event(event)

            if widget_event:
                return widget_event

        return None

    def draw(self):
        sprites = self.sprites()
        for spr in sprites:
            surface = spr._surface or window.screen
            self.spritedict[spr] = surface.blit(spr.image, spr.rect)
        self.lostsprites = []
        dirty = self.lostsprites

        return dirty

    def on_widget(self, mouse_pos):
        test_sprite = pygame.sprite.Sprite()
        test_sprite.rect = pygame.FRect(*mouse_pos, 1, 1)
        return pygame.sprite.spritecollideany(test_sprite, self)
```

```python
import sqlite3
from pathlib import Path

database_path = (Path(__file__).parent / '../database.db').resolve()

def upgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games ADD COLUMN created_dt TIMESTAMP NOT NULL
    ''')

    connection.commit()
    connection.close()

def downgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games DROP COLUMN created_dt
    ''')

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

```python
import sqlite3
from pathlib import Path

database_path = (Path(__file__).parent / '../database.db').resolve()

```

```
def upgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games ADD COLUMN fen_string TEXT NOT NULL
    ''')

    connection.commit()
    connection.close()

def downgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games DROP COLUMN fen_string
    ''')

    connection.commit()
    connection.close()

upgrade()
```

```
import sqlite3
from pathlib import Path

database_path = (Path(__file__).parent / '../database.db').resolve()

def upgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games ADD COLUMN start_fen_string TEXT NOT NULL
    ''')

    connection.commit()
    connection.close()

def downgrade():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games DROP COLUMN start_fen_string
    ''')

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

```
import sqlite3
from pathlib import Path

database_path = (Path(__file__).parent / '../database.db').resolve()

def upgrade():
    """
    Upgrade function to rename fen_string column.
```

```python
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games RENAME COLUMN fen_string TO final_fen_string
    ''')

    connection.commit()
    connection.close()

def downgrade():
    """
    Downgrade function to revert fen_string column renaming.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        ALTER TABLE games RENAME COLUMN final_fen_string TO fen_string
    ''')

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

```python
import sqlite3
from pathlib import Path

database_path = (Path(__file__).parent / '../database.db').resolve()

def upgrade():
    """
    Upgrade function to create games table.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        CREATE TABLE games(
            id INTEGER PRIMARY KEY,
            cpu_enabled INTEGER NOT NULL,
            cpu_depth INTEGER,
            winner INTEGER,
            time_enabled INTEGER NOT NULL,
            time REAL,
            number_of_ply INTEGER NOT NULL,
            moves TEXT NOT NULL
        )
    ''')

    connection.commit()
    connection.close()

def downgrade():
    """
    Downgrade function to revert table creation.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        DROP TABLE games
    ''')

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

```python
import pygame
from data.utils.asset_helpers import scale_and_cache

FPS = 60

class AnimationManager:
    def __init__(self):
        self._current_ms = 0
        self._timers = []

    def set_delta_time(self):
        self._current_ms = pygame.time.get_ticks()

        for timer in self._timers:
            start_ms, target_ms, callback = timer
            if self._current_ms - start_ms >= target_ms:
                callback()
                self._timers.remove(timer)

    def calculate_frame_index(self, start_index, end_index, fps):
        ms_per_frame = int(1000 / fps)
        return start_index + ((self._current_ms // ms_per_frame) % (end_index - start_index))

    def draw_animation(self, screen, animation, position, size, fps=8):
        frame_index = self.calculate_frame_index(0, len(animation), fps)
        scaled_animation = scale_and_cache(animation[frame_index], size)
        screen.blit(scaled_animation, position)

    def draw_image(self, screen, image, position, size):
        scaled_background = scale_and_cache(image, size)
        screen.blit(scaled_background, position)

    def set_timer(self, target_ms, callback):
        self._timers.append((self._current_ms, target_ms, callback))

animation = AnimationManager()
```

```python
import pygame
from data.utils.data_helpers import get_user_settings
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)
user_settings = get_user_settings()

class AudioManager:
    def __init__(self, num_channels=16):
        pygame.mixer.set_num_channels(num_channels)

        self._music_volume = user_settings['musicVolume']
        self._sfx_volume = user_settings['sfxVolume']

        self._current_song = None
        self._current_channels = []

    def set_sfx_volume(self, volume):
        self._sfx_volume = volume

        for channel in self._current_channels:
            channel.set_volume(self._sfx_volume)

    def set_music_volume(self, volume):
        self._music_volume = volume

        pygame.mixer.music.set_volume(self._music_volume)

    def pause_sfx(self):
        pygame.mixer.pause()

    def unpause_sfx(self):
        pygame.mixer.unpause()

    def stop_sfx(self, fadeout=0):
        pygame.mixer.fadeout(fadeout)

    def remove_unused_channels(self):
        unused_channels = []
        for channel in self._current_channels:
            if channel.get_busy() is False:
                unused_channels.append(channel)

        return unused_channels

    def play_sfx(self, sfx, loop=False):
        unused_channels = self.remove_unused_channels()

        if len(unused_channels) == 0:
            channel = pygame.mixer.find_channel()
        else:
            channel = unused_channels.pop(0)

        if channel is None:
            logger.warning('No available channel for SFX')
            return

        self._current_channels.append(channel)
        channel.set_volume(self._sfx_volume)

        if loop:
            channel.play(sfx, loops=-1)
        else:
            channel.play(sfx)

    def play_music(self, music_path):
        if 'menu' in str(music_path) and 'menu' in str(self._current_song):
            return

        if music_path == self._current_song:
            return

        pygame.mixer.music.stop()
        pygame.mixer.music.unload()
        pygame.mixer.music.load(music_path)
        pygame.mixer.music.set_volume(self._music_volume)
        pygame.mixer.music.play(loops=-1)

        self._current_song = music_path

audio = AudioManager()
```

```python
import pygame
from data.assets import GRAPHICS
from data.constants import CursorMode

class CursorManager:
    def __init__(self):
        self._mode = CursorMode.ARROW
        self.set_mode(CursorMode.ARROW)

    def set_mode(self, mode):
        pygame.mouse.set_visible(True)

        match mode:
            case CursorMode.ARROW:
                pygame.mouse.set_cursor((7, 5), pygame.transform.scale(GRAPHICS['arrow'], (32, 32)))
            case CursorMode.IBEAM:
                pygame.mouse.set_cursor((15, 5), pygame.transform.scale(GRAPHICS['ibeam'], (32, 32)))
            case CursorMode.OPENHAND:
                pygame.mouse.set_cursor((17, 5), pygame.transform.scale(GRAPHICS['hand_open'], (32, 32)))
            case CursorMode.CLOSEDHAND:
                pygame.mouse.set_cursor((17, 5), pygame.transform.scale(GRAPHICS['hand_closed'], (32, 32)))
            case CursorMode.NO:
                pygame.mouse.set_visible(False)

        self._mode = mode

    def get_mode(self):
        return self._mode

cursor = CursorManager()
```

```python
import logging.config
from data.utils.data_helpers import load_json
from pathlib import Path
import logging

config_path = (Path(__file__).parent / '../app_data/logs_config.json').resolve()
config = load_json(config_path)
logging.config.dictConfig(config)

def initialise_logger(file_path):
    return logging.getLogger(Path(file_path).name)
```

```python
from pathlib import Path
from array import array
import moderngl
from data.shaders.classes import shader_pass_lookup
from data.shaders.protocol import SMProtocol
from data.constants import ShaderType

```

```python
shader_path = (Path(__file__).parent / '../shaders/').resolve()

SHADER_PRIORITY = [
    ShaderType.CRT,
    ShaderType.SHAKE,
    ShaderType.BLOOM,
    ShaderType.CHROMATIC_ABBREVIATION,
    ShaderType.RAYS,
    ShaderType.GRAYSCALE,
    ShaderType.BASE,
]

pygame_quad_array = array('f', [
    -1.0, 1.0, 0.0, 0.0,
    1.0, 1.0, 1.0, 0.0,
    -1.0, -1.0, 0.0, 1.0,
    1.0, -1.0, 1.0, 1.0,
])

opengl_quad_array = array('f', [
    -1.0, -1.0, 0.0, 0.0,
    1.0, -1.0, 1.0, 0.0,
    -1.0, 1.0, 0.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
])

class ShaderManager(SMProtocol):
    def __init__(self, ctx: moderngl.Context, screen_size):
        self._ctx = ctx
        self._ctx.gc_mode = 'auto'

        self._screen_size = screen_size
        self._opengl_buffer = self._ctx.buffer(data=opengl_quad_array)
        self._pygame_buffer = self._ctx.buffer(data=pygame_quad_array)
        self._shader_list = [ShaderType.BASE]

        self._vert_shaders = {}
        self._frag_shaders = {}
        self._programs = {}
        self._vaos = {}
        self._textures = {}
        self._shader_passes = {}
        self.framebuffers = {}

        self.load_shader(ShaderType.BASE)
        self.load_shader(ShaderType._CALIBRATE)
        self.create_framebuffer(ShaderType._CALIBRATE)

    def load_shader(self, shader_type, **kwargs):
        """
        Loads a given shader by creating a VAO reading the corresponding .frag file.

        Args:
            shader_type (ShaderType): The type of shader to load.
            **kwargs: Additional arguments passed when initialising the fragment shader class.
        """
        self._shader_passes[shader_type] = shader_pass_lookup[shader_type](self, **kwargs)
        self.create_vao(shader_type)

    def clear_shaders(self):
        """
        Clears the shader list, leaving only the base shader.
        """
        self._shader_list = [ShaderType.BASE]

    def create_vao(self, shader_type):
        """
        Creates a vertex array object (VAO) for the given shader type.

        Args:
            shader_type (ShaderType): The type of shader.
        """
        frag_name = shader_type[1:] if shader_type[0] == '_' else shader_type
        vert_path = Path(shader_path / 'vertex/base.vert').resolve()
        frag_path = Path(shader_path / f'fragments/{frag_name}.frag').resolve()

        self._vert_shaders[shader_type] = vert_path.read_text()
        self._frag_shaders[shader_type] = frag_path.read_text()

        program = self._ctx.program(vertex_shader=self._vert_shaders[shader_type],
     fragment_shader=self._frag_shaders[shader_type])
        self._programs[shader_type] = program

        if shader_type == ShaderType._CALIBRATE:
            self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._pygame_buffer, '2f 2f', 'vert', 'texCoords')])
        else:
            self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._opengl_buffer, '2f 2f', 'vert', 'texCoords')])

    def create_framebuffer(self, shader_type, size=None, filter=moderngl.NEAREST):
        """
        Creates a framebuffer for the given shader type.

        Args:
            shader_type (ShaderType): The type of shader.
            size (tuple[int, int], optional): The size of the framebuffer.
    Defaults to screen size.
            filter (moderngl.Filter, optional): The texture filter. Defaults to
    NEAREST.
        """
        texture_size = size or self._screen_size
        texture = self._ctx.texture(size=texture_size, components=4)
        texture.filter = (filter, filter)

        self._textures[shader_type] = texture
        self.framebuffers[shader_type] = self._ctx.framebuffer(color_attachments=[
    self._textures[shader_type]])

    def render_to_fbo(self, shader_type, texture, output_fbo=None, program_type=
    None, use_image=True, **kwargs):
        """
        Applies the shaders and renders the resultant texture to a framebuffer
    object (FBO).

        Args:
            shader_type (ShaderType): The type of shader.
            texture (moderngl.Texture): The texture to render.
            output_fbo (moderngl.Framebuffer, optional): The output framebuffer.
    Defaults to None.
            program_type (ShaderType, optional): The program type. Defaults to
```

```python
    None.
            use_image (bool, optional): Whether to use the image uniform. Defaults to True.
            **kwargs: Additional uniforms for the fragment shader.
        """
        fbo = output_fbo or self.framebuffers[shader_type]
        program = self._programs[program_type] if program_type else self._programs[shader_type]
        vao= self._vaos[program_type] if program_type else self._vaos[shader_type]

        fbo.use()
        texture.use(0)

        if use_image:
            program['image'] = 0
        for uniform, value in kwargs.items():
            program[uniform] = value

        vao.render(mode=moderngl.TRIANGLE_STRIP)

    def apply_shader(self, shader_type, **kwargs):
        """
        Applies a shader of the given type and adds it to the list.

        Args:
            shader_type (ShaderType): The type of shader to apply.

        Raises:
            ValueError: If the shader is already being applied.
        """
        if shader_type in self._shader_list:
            return

        self.load_shader(shader_type, **kwargs)
        self._shader_list.append(shader_type)

        # Sort shader list based on the order in SHADER_PRIORITY, so that more important shaders are applied first
        self._shader_list.sort(key=lambda shader: -SHADER_PRIORITY.index(shader))

    def remove_shader(self, shader_type):
        """
        Removes a shader of the given type from the list.

        Args:
            shader_type (ShaderType): The type of shader to remove.
        """
        if shader_type in self._shader_list:
            self._shader_list.remove(shader_type)

    def render_output(self):
        """
        Renders the final output to the screen.
        """
        # Render to the screen framebuffer
        self._ctx.screen.use()

        # Take the texture of the last framebuffer to be rendered to, and render that to the screen framebuffer
        output_shader_type = self._shader_list[-1]
        self.get_fbo_texture(output_shader_type).use(0)
        self._programs[output_shader_type]['image'] = 0

        self._vaos[output_shader_type].render(mode=moderngl.TRIANGLE_STRIP)

    def get_fbo_texture(self, shader_type):
        """
        Gets the texture from the specified shader type's FBO.

        Args:
            shader_type (ShaderType): The type of shader.

        Returns:
            moderngl.Texture: The texture from the FBO.
        """
        return self.framebuffers[shader_type].color_attachments[0]

    def calibrate_pygame_surface(self, pygame_surface):
        """
        Converts the Pygame window surface into an OpenGL texture.

        Args:
            pygame_surface (pygame.Surface): The finished Pygame surface.

        Returns:
            moderngl.Texture: The calibrated texture.
        """
        texture = self._ctx.texture(pygame_surface.size, 4)
        texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
        texture.swizzle = 'BGRA'
        # Take the Pygame surface's pixel array and draw it to the new texture
        texture.write(pygame_surface.get_view('1'))

        # ShaderType._CALIBRATE has a VAO containing the pygame_quad_array coordinates, as Pygame uses different texture coordinates than ModernGL textures
        self.render_to_fbo(ShaderType._CALIBRATE, texture)
        return self.get_fbo_texture(ShaderType._CALIBRATE)

    def draw(self, surface, arguments):
        """
        Draws the Pygame surface with shaders applied to the screen.

        Args:
            surface (pygame.Surface): The final Pygame surface.
            arguments (dict): A dict of { ShaderType: Args } items, containing keyword arguments for every fragment shader.
        """
        self._ctx.viewport = (0, 0, *self._screen_size)
        texture = self.calibrate_pygame_surface(surface)

        for shader_type in self._shader_list:
            self._shader_passes[shader_type].apply(texture, **arguments.get(shader_type, {}))
            texture = self.get_fbo_texture(shader_type)

        self.render_output()

    def __del__(self):
        """
        Cleans up ModernGL resources when the ShaderManager object is deleted.
        """
        self.cleanup()

    def cleanup(self):
        """
        Cleans up resources used by the ModernGL.
        Probably unnecessary as the 'auto' garbage collection mode is used.
        """
        self._pygame_buffer.release()
        self._opengl_buffer.release()
        for program in self._programs:
            self._programs[program].release()
        for texture in self._textures:
            self._textures[texture].release()
        for vao in self._vaos:
            self._vaos[vao].release()
        for framebuffer in self.framebuffers:
            self.framebuffers[framebuffer].release()

    def handle_resize(self, new_screen_size):
        """
        Handles resizing of the screen.

        Args:
            new_screen_size (tuple[int, int]): The new screen size.
        """
        self._screen_size = new_screen_size

        # Recreate all framebuffers to prevent scaling issues
        for shader_type in self.framebuffers:
            filter = self._textures[shader_type].filter[0]
            self.create_framebuffer(shader_type, size=self._screen_size, filter=
    filter)
```

```python
from data.utils.data_helpers import get_themes, get_user_settings

themes = get_themes()
user_settings = get_user_settings()

def flatten_dictionary_generator(dictionary, parent_key=None):
    """
    Recursive depth-first search to yield all items in a dictionary.

    Args:
        dictionary (dict): Dictionary to be iterated through.
        parent_key (str, optional): Prefix added to every key. Defaults to None.

    Yields:
        dict | tuple[str, str]: Another dictionary or key, value pair.
    """
    for key, value in dictionary.items():
        if parent_key:
            new_key = parent_key + key.capitalize()
        else:
            new_key = key

        if isinstance(value, dict):
            yield from flatten_dictionary(value, new_key).items()
        else:
            yield new_key, value

def flatten_dictionary(dictionary, parent_key=''):
    return dict(flatten_dictionary_generator(dictionary, parent_key))

class ThemeManager:
    def __init__(self):
        self.__dict__.update(flatten_dictionary(themes['colours']))
        self.__dict__.update(flatten_dictionary(themes['dimensions']))

    def __getitem__(self, arg):
        """
        Override default class's __getitem__ dunder method, to make retrieving an instance attribute nicer with [] notation.

        Args:
            arg (str): Attribute name.

        Raises:
            KeyError: Instance does not have requested attribute.

        Returns:
            str | int: Instance attribute.
        """
        item = self.__dict__.get(arg)

        if item is None:
            raise KeyError('(ThemeManager.__getitem__) Requested theme item not found:', arg)

        return item

theme = ThemeManager()
```

```python
import pygame
import moderngl
from data.constants import ShaderType, SCREEN_SIZE, SHADER_MAP
from data.utils.data_helpers import get_user_settings
from data.utils.asset_helpers import draw_background
from data.managers.shader import ShaderManager

user_settings = get_user_settings()
is_opengl = user_settings['opengl']
is_fullscreen = user_settings['displayMode'] == 'fullscreen'

class WindowManager(pygame.Window):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self._native_screen = self.get_surface() # Initialise convert format
        self.screen = pygame.Surface(self.size, pygame.SRCALPHA)

        if is_opengl:
            self._ctx = moderngl.create_context()
            self._shader_manager = ShaderManager(self._ctx, screen_size=self.size)

            self.shader_arguments = {
                ShaderType.BASE: {},
                ShaderType.SHAKE: {},
                ShaderType.BLOOM: {},
                ShaderType.GRAYSCALE: {},
                ShaderType.CRT: {},
                ShaderType.RAYS: {}
            }

            if (selected_shader := get_user_settings()['shader']) is not None:
                for shader_type in SHADER_MAP[selected_shader]:
                    self.set_effect(shader_type)
        else:
            from data.assets import GRAPHICS
            self._background_image = GRAPHICS['temp_background']

    def set_effect(self, effect, **kwargs):
        if is_opengl:
            self._shader_manager.apply_shader(effect, **kwargs)

    def set_apply_arguments(self, effect, **kwargs):
        if is_opengl:
            self.shader_arguments[effect] = kwargs

    def clear_apply_arguments(self, effect):
        if is_opengl:
            self.shader_arguments[effect] = {}

    def clear_effect(self, effect):
        if is_opengl:
            self._shader_manager.remove_shader(effect)
            self.clear_apply_arguments(effect)

    def clear_all_effects(self, clear_arguments=False):
        if is_opengl:
            self._shader_manager.clear_shaders()

            if clear_arguments:
                for shader_type in self.shader_arguments:
                    self.shader_arguments[shader_type] = {}

    def draw(self):
        if is_opengl:
            self._shader_manager.draw(self.screen, self.shader_arguments)
        else:
            self._native_screen.blit(self.screen, (0, 0))

        self.flip()

        if is_opengl:
            self.screen.fill((0, 0, 0, 0))
        else:
            self.screen.fill((0, 0, 0))
            draw_background(self.screen, self._background_image)

    def update(self):
        self.draw()

    def handle_resize(self):
        self.screen = pygame.Surface(self.size, pygame.SRCALPHA)
        if is_opengl:
            self._shader_manager.handle_resize(self.size)
        else:
            draw_background(self.screen, self._background_image)

window = WindowManager(size=SCREEN_SIZE, resizable=True, opengl=is_opengl,
    fullscreen_desktop=is_fullscreen)
```

```
import pygame
import moderngl
from typing import Protocol, Optional
from data.constants import ShaderType

class SMProtocol(Protocol):
    def load_shader(self, shader_type: ShaderType, **kwargs) -> None: ...
```

```python
    def clear_shaders(self) -> None: ...
    def create_vao(self, shader_type: ShaderType) -> None: ...
    def create_framebuffer(self, shader_type: ShaderType, size: Optional[tuple[int]]=None, filter: Optional[int]=moderngl.NEAREST) -> None: ...
    def render_to_fbo(self, shader_type: ShaderType, texture: moderngl.Texture, output_fbo: Optional[moderngl.Framebuffer] = None, program_type: Optional[ShaderType] = None, use_image: Optional[bool] = True, **kwargs) -> None: ...
    def apply_shader(self, shader_type: ShaderType, **kwargs) -> None: ...
    def remove_shader(self, shader_type: ShaderType) -> None: ...
    def render_output(self, texture: moderngl.Texture) -> None: ...
    def get_fbo_texture(self, shader_type: ShaderType) -> moderngl.Texture: ...
    def calibrate_pygame_surface(self, pygame_surface: pygame.Surface) -> moderngl.Texture: ...
    def draw(self, surface: pygame.Surface, arguments: dict) -> None: ...
    def __del__(self) -> None: ...
    def cleanup(self) -> None: ...
    def handle_resize(self, new_screen_size: tuple[int]) -> None: ...

    _ctx: moderngl.Context
    _screen_size: tuple[int]
    _opengl_buffer: moderngl.Buffer
    _pygame_buffer: moderngl.Buffer
    _shader_stack: list[ShaderType]

    _vert_shaders: dict
    _frag_shaders: dict
    _programs: dict
    _vaos: dict
    _textures: dict
    _shader_passes: dict
    framebuffers: dict
```

```python
import pygame
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class Base:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        self._shader_manager.create_framebuffer(ShaderType.BASE)
        self._shader_manager.create_vao(ShaderType.BACKGROUND_WAVES)
        self._shader_manager.create_vao(ShaderType.BACKGROUND_BALATRO)
        self._shader_manager.create_vao(ShaderType.BACKGROUND_LASERS)
        self._shader_manager.create_vao(ShaderType.BACKGROUND_GRADIENT)
        self._shader_manager.create_vao(ShaderType.BACKGROUND_NONE)

    def apply(self, texture, background_type=None):
        base_texture = self._shader_manager.get_fbo_texture(ShaderType.BASE)

        match background_type:
            case ShaderType.BACKGROUND_WAVES:
                self._shader_manager.render_to_fbo(
                    ShaderType.BASE,
                    texture=base_texture,
                    program_type=ShaderType.BACKGROUND_WAVES,
                    use_image=False,
                    time=pygame.time.get_ticks() / 1000
                )
            case ShaderType.BACKGROUND_BALATRO:
                self._shader_manager.render_to_fbo(
                    ShaderType.BASE,
                    texture=base_texture,
                    program_type=ShaderType.BACKGROUND_BALATRO,
                    use_image=False,
                    time=pygame.time.get_ticks() / 1000,
                    screenSize=base_texture.size
                )
            case ShaderType.BACKGROUND_LASERS:
                self._shader_manager.render_to_fbo(
                    ShaderType.BASE,
                    texture=base_texture,
                    program_type=ShaderType.BACKGROUND_LASERS,
                    use_image=False,
                    time=pygame.time.get_ticks() / 1000,
                   screenSize=base_texture.size
                )
            case ShaderType.BACKGROUND_GRADIENT:
                self._shader_manager.render_to_fbo(
                    ShaderType.BASE,
                    texture=base_texture,
                    program_type=ShaderType.BACKGROUND_GRADIENT,
                    use_image=False,
                    time=pygame.time.get_ticks() / 1000,
                   screenSize=base_texture.size
                )
            case None:
                self._shader_manager.render_to_fbo(
                    ShaderType.BASE,
                    texture=base_texture,
                    program_type=ShaderType.BACKGROUND_NONE,
                    use_image=False,
                )
            case _:
                raise ValueError('(shader.py) Unknown background type:', background_type)

        self._shader_manager.get_fbo_texture(ShaderType.BASE).use(1)
        self._shader_manager.render_to_fbo(ShaderType.BASE, texture, background=1)
```

```
import moderngl
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class _Blend:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        self._shader_manager.create_framebuffer(ShaderType._BLEND)

    def apply(self, texture, texture_2, texture_2_pos):
        self._shader_manager._ctx.blend_func = (moderngl.SRC_ALPHA, moderngl.ONE)

        relative_size = (texture_2.size[0] / texture.size[0], texture_2.size[1] / texture.size[1])
        opengl_pos = (texture_2_pos[0], 1 - texture_2_pos[1] - relative_size[1])

        texture_2.use(1)
        self._shader_manager.render_to_fbo(ShaderType._BLEND, texture, image2=1, image2Pos=opengl_pos, relativeSize=relative_size)
        self._shader_manager._ctx.blend_func = moderngl.DEFAULT_BLENDING
```

```
from data.shaders.classes.highlight_brightness import _HighlightBrightness
from data.shaders.classes.highlight_colour import _HighlightColour
```

```python
from data.shaders.protocol import SMProtocol
from data.shaders.classes.blur import _Blur
from data.constants import ShaderType

BLOOM_INTENSITY = 0.6

class Bloom:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.load_shader(ShaderType._BLUR)
        shader_manager.load_shader(ShaderType._HIGHLIGHT_BRIGHTNESS)
        shader_manager.load_shader(ShaderType._HIGHLIGHT_COLOUR)

        shader_manager.create_framebuffer(ShaderType.BLOOM)
        shader_manager.create_framebuffer(ShaderType._BLUR)
        shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_BRIGHTNESS)
        shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_COLOUR)

    def apply(self, texture, highlight_surface=None, highlight_colours=[],
    surface_intensity=BLOOM_INTENSITY, brightness_intensity=BLOOM_INTENSITY,
    colour_intensity=BLOOM_INTENSITY):
        """
        Applies a bloom effect to a given texture.

        Args:
            texture (moderngl.Texture): Texture to apply bloom to.
            highlight_surface (pygame.Surface, optional): Surface to use as the
    highlights. Defaults to None.
            highlight_colours (list[list[int, int, int], ...], optional): Colours
    to use as the highlights. Defaults to [].
            surface_intensity (_type_, optional): Intensity of bloom applied to
    the highlight surface. Defaults to BLOOM_INTENSITY.
            brightness_intensity (_type_, optional): Intensity of bloom applied to
     the highlight brightness. Defaults to BLOOM_INTENSITY.
            colour_intensity (_type_, optional): Intensity of bloom applied to the
     highlight colours. Defaults to BLOOM_INTENSITY.
        """
        if highlight_surface:
            # Calibrate Pygame surface and apply blur
            glare_texture = self._shader_manager.calibrate_pygame_surface(
    highlight_surface)
            _Blur(self._shader_manager).apply(glare_texture)

            self._shader_manager.get_fbo_texture(ShaderType._BLUR).use(1)
            self._shader_manager.render_to_fbo(ShaderType.BLOOM, texture,
    blurredImage=1, intensity=surface_intensity)

            # Set bloom-applied texture as the base texture
            texture = self._shader_manager.get_fbo_texture(ShaderType.BLOOM)

        # Extract bright colours (highlights) from the texture
        _HighlightBrightness(self._shader_manager).apply(texture, intensity=
    brightness_intensity)
        highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
    _HIGHLIGHT_BRIGHTNESS)

        # Use colour as highlights
        for colour in highlight_colours:
            _HighlightColour(self._shader_manager).apply(texture, old_highlight=
    highlight_texture, colour=colour, intensity=colour_intensity)
            highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
    _HIGHLIGHT_COLOUR)

        # Apply Gaussian blur to highlights
        _Blur(self._shader_manager).apply(highlight_texture)

        # Add the pixel values for the highlights onto the base texture
        self._shader_manager.get_fbo_texture(ShaderType._BLUR).use(1)
        self._shader_manager.render_to_fbo(ShaderType.BLOOM, texture, blurredImage
    =1, intensity=BLOOM_INTENSITY)
```

```
from data.shaders.protocol import SMProtocol
from data.constants import ShaderType

BLUR_ITERATIONS = 4

class _Blur:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.create_framebuffer(ShaderType._BLUR)

        shader_manager.create_framebuffer("blurPing")
        shader_manager.create_framebuffer("blurPong")

    def apply(self, texture):
        """
        Applies Gaussian blur to a given texture.

        Args:
            texture (moderngl.Texture): Texture to blur.
        """
        self._shader_manager.get_fbo_texture("blurPong").write(texture.read())

        for _ in range(BLUR_ITERATIONS):
            # Apply horizontal blur
            self._shader_manager.render_to_fbo(
                ShaderType._BLUR,
                texture=self._shader_manager.get_fbo_texture("blurPong"),
                output_fbo=self._shader_manager.framebuffers["blurPing"],
                passes=5,
                horizontal=True
            )
            # Apply vertical blur
            self._shader_manager.render_to_fbo(
                ShaderType._BLUR,
                texture=self._shader_manager.get_fbo_texture("blurPing"), # Use
    horizontal blur result as input texture
                output_fbo=self._shader_manager.framebuffers["blurPong"],
                passes=5,
                horizontal=False
            )

        self._shader_manager.render_to_fbo(ShaderType._BLUR, self._shader_manager.
    get_fbo_texture("blurPong"))
```

```
import pygame
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

CHROMATIC_ABBREVIATION_INTENSITY = 2.0

class ChromaticAbbreviation:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        self._shader_manager.create_framebuffer(ShaderType.CHROMATIC_ABBREVIATION)

    def apply(self, texture):
        mouse_pos = (pygame.mouse.get_pos()[0] / texture.size[0], pygame.mouse.get_pos()[1] / texture.size[1])
        self._shader_manager.render_to_fbo(ShaderType.CHROMATIC_ABBREVIATION, texture, mouseFocusPoint=mouse_pos, enabled=pygame.mouse.get_pressed()[0], intensity=CHROMATIC_ABBREVIATION_INTENSITY)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class _Crop:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

    def apply(self, texture, relative_pos, relative_size):
        opengl_pos = (relative_pos[0], 1 - relative_pos[1] - relative_size[1])
        pixel_size = (int(relative_size[0] * texture.size[0]), int(relative_size[1] * texture.size[1]))

        self._shader_manager.create_framebuffer(ShaderType._CROP, size=pixel_size)

        self._shader_manager.render_to_fbo(ShaderType._CROP, texture, relativePos=opengl_pos, relativeSize=relative_size)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class CRT:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.create_framebuffer(ShaderType.CRT)

    def apply(self, texture):
        self._shader_manager.render_to_fbo(ShaderType.CRT, texture)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class Grayscale:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.create_framebuffer(ShaderType.GRAYSCALE)

    def apply(self, texture):
        self._shader_manager.render_to_fbo(ShaderType.GRAYSCALE, texture)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

HIGHLIGHT_THRESHOLD = 0.9

class _HighlightBrightness:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_BRIGHTNESS)

    def apply(self, texture, intensity):
        self._shader_manager.render_to_fbo(ShaderType._HIGHLIGHT_BRIGHTNESS, texture, threshold=HIGHLIGHT_THRESHOLD, intensity=intensity)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class _HighlightColour:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_COLOUR)

    def apply(self, texture, old_highlight, colour, intensity):
        old_highlight.use(1)
        self._shader_manager.render_to_fbo(ShaderType._HIGHLIGHT_COLOUR, texture, highlight=1, colour=colour, threshold=0.1, intensity=intensity)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol
from data.shaders.classes.shadowmap import _Shadowmap

LIGHT_RESOLUTION = 256

class _Lightmap:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.load_shader(ShaderType._SHADOWMAP)

    def apply(self, texture, colour, softShadow, occlusion=None, falloff=0.0, clamp=(-180, 180)):
        self._shader_manager.create_framebuffer(ShaderType._LIGHTMAP, size=texture.size)
        self._shader_manager._ctx.enable(self._shader_manager._ctx.BLEND)

        _Shadowmap(self._shader_manager).apply(texture, occlusion)
        shadow_map = self._shader_manager.get_fbo_texture(ShaderType._SHADOWMAP)

        self._shader_manager.render_to_fbo(ShaderType._LIGHTMAP, shadow_map, resolution=LIGHT_RESOLUTION, lightColour=colour, falloff=falloff, angleClamp=clamp, softShadow=softShadow)

        self._shader_manager._ctx.disable(self._shader_manager._ctx.BLEND)
```

```
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol

class _Occlusion:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

    def apply(self, texture, occlusion_colour=(255, 0, 0)):
        self._shader_manager.create_framebuffer(ShaderType._OCCLUSION, size=texture.size)
        self._shader_manager.render_to_fbo(ShaderType._OCCLUSION, texture, checkColour=tuple(num / 255 for num in occlusion_colour))
```

```python
from data.shaders.classes.lightmap import _Lightmap
from data.shaders.classes.blend import _Blend
from data.shaders.protocol import SMProtocol
from data.shaders.classes.crop import _Crop
from data.constants import ShaderType

class Rays:
    def __init__(self, shader_manager: SMProtocol, lights):
        self._shader_manager = shader_manager
        self._lights = lights

        # Load all necessary shaders
        shader_manager.load_shader(ShaderType._LIGHTMAP)
        shader_manager.load_shader(ShaderType._BLEND)
        shader_manager.load_shader(ShaderType._CROP)
        shader_manager.create_framebuffer(ShaderType.RAYS)

    def apply(self, texture, occlusion=None, softShadow=0.3):
        """
        Applies the light rays effect to a given texture.

        Args:
            texture (moderngl.Texture): The texture to apply the effect to.
            occlusion (pygame.Surface, optional): A Pygame mask surface to use as the occlusion texture. Defaults to None.
        """
        final_texture = texture

        # Iterate through array containing light information
        for pos, radius, colour, *args in self._lights:
            # Topleft of light source square
            light_topleft = (pos[0] - (radius * texture.size[1] / texture.size[0]), pos[1] - radius)
            # Relative size of light compared to texture
            relative_size = (radius * 2 * texture.size[1] / texture.size[0], radius * 2)

            # Crop texture to light source diameter, and to position light source at the center
            _Crop(self._shader_manager).apply(texture, relative_pos=light_topleft, relative_size=relative_size)
            cropped_texture = self._shader_manager.get_fbo_texture(ShaderType._CROP)

            if occlusion:
                # Calibrate Pygame mask surface and crop it
                occlusion_texture = self._shader_manager.calibrate_pygame_surface(occlusion)
                _Crop(self._shader_manager).apply(occlusion_texture, relative_pos=light_topleft, relative_size=relative_size)
                occlusion_texture = self._shader_manager.get_fbo_texture(ShaderType._CROP)
            else:
                occlusion_texture = None

            # Apply lightmap shader, shadowmap and occlusion are included within the _Lightmap class
            _Lightmap(self._shader_manager).apply(cropped_texture, colour, softShadow, occlusion_texture, *args)
            light_map = self._shader_manager.get_fbo_texture(ShaderType._LIGHTMAP)

            # Blend the final result with the original texture
            _Blend(self._shader_manager).apply(final_texture, light_map, light_topleft)
            final_texture = self._shader_manager.get_fbo_texture(ShaderType._BLEND)

        self._shader_manager.render_to_fbo(ShaderType.RAYS, final_texture)
```

```python
import moderngl
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol
from data.shaders.classes.occlusion import _Occlusion

LIGHT_RESOLUTION = 256

class _Shadowmap:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        shader_manager.load_shader(ShaderType._OCCLUSION)

    def apply(self, texture, occlusion_texture=None):
        self._shader_manager.create_framebuffer(ShaderType._SHADOWMAP, size=(texture.size[0], 1), filter=moderngl.LINEAR)

        if occlusion_texture is None:
            _Occlusion(self._shader_manager).apply(texture)
            occlusion_texture = self._shader_manager.get_fbo_texture(ShaderType._OCCLUSION)

        self._shader_manager.render_to_fbo(ShaderType._SHADOWMAP, occlusion_texture, resolution=LIGHT_RESOLUTION)
```

```python
from data.constants import ShaderType
from data.shaders.protocol import SMProtocol
from random import randint

SHAKE_INTENSITY = 3

class Shake:
    def __init__(self, shader_manager: SMProtocol):
        self._shader_manager = shader_manager

        self._shader_manager.create_framebuffer(ShaderType.SHAKE)

    def apply(self, texture, intensity=SHAKE_INTENSITY):
        displacement = (randint(-intensity, intensity) / 1000, randint(-intensity, intensity) / 1000)
        self._shader_manager.render_to_fbo(ShaderType.SHAKE, texture, displacement=displacement)
```

```python
from data.shaders.classes.chromatic_abbreviation import ChromaticAbbreviation
from data.shaders.classes.highlight_brightness import _HighlightBrightness
from data.shaders.classes.highlight_colour import _HighlightColour
from data.shaders.classes.shadowmap import _Shadowmap
from data.shaders.classes.occlusion import _Occlusion
from data.shaders.classes.grayscale import Grayscale
from data.shaders.classes.lightmap import _Lightmap
from data.shaders.classes.blend import _Blend
from data.shaders.classes.shake import Shake
from data.shaders.classes.bloom import Bloom
from data.shaders.classes.blur import _Blur
from data.shaders.classes.crop import _Crop
from data.shaders.classes.rays import Rays
from data.shaders.classes.base import Base
from data.shaders.classes.crt import CRT
from data.constants import ShaderType

shader_pass_lookup = {
    ShaderType.CHROMATIC_ABBREVIATION: ChromaticAbbreviation,
    ShaderType.GRAYSCALE: Grayscale,
    ShaderType.SHAKE: Shake,
    ShaderType.BLOOM: Bloom,
    ShaderType.BASE: Base,
    ShaderType.RAYS: Rays,
    ShaderType.CRT: CRT,

    ShaderType._HIGHLIGHT_BRIGHTNESS: _HighlightBrightness,
    ShaderType._HIGHLIGHT_COLOUR: _HighlightColour,
    ShaderType._CALIBRATE: lambda *args: None,
    ShaderType._OCCLUSION: _Occlusion,
    ShaderType._SHADOWMAP: _Shadowmap,
    ShaderType._LIGHTMAP: _Lightmap,
    ShaderType._BLEND: _Blend,
    ShaderType._BLUR: _Blur,
    ShaderType._CROP: _Crop,
}
```

```python
import pygame
import pyperclip
from data.constants import BrowserEventType, ShaderType, GAMES_PER_PAGE
from data.utils.database_helpers import delete_game, get_ordered_games
from data.states.browser.widget_dict import BROWSER_WIDGETS
from data.managers.logs import initialise_logger
from data.managers.window import window
from data.control import _State
from data.assets import MUSIC
from random import randint

logger = initialise_logger(__name__)

class Browser(_State):
    def __init__(self):
        super().__init__()

        self._selected_index = None
        self._filter_column = 'number_of_ply'
        self._filter_ascend = False
        self._games_list = []
        self._page_number = 1

    def cleanup(self):
        super().cleanup()

        if self._selected_index is not None:
            return self._games_list[self._selected_index]

        return None

    def startup(self, persist=None):
        self.refresh_games_list() # BEFORE RESIZE TO FILL WIDGET BEFORE RESIZING
        super().startup(BROWSER_WIDGETS, music=MUSIC[f'menu_{randint(1, 3)}'])

        self._filter_column = 'number_of_ply'
        self._filter_ascend = False

        window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.BACKGROUND_BALATRO)

        BROWSER_WIDGETS['help'].kill()
        BROWSER_WIDGETS['browser_strip'].kill()

        self.draw()

    def refresh_games_list(self):
        column_map = {
            'moves': 'number_of_ply',
            'winner': 'winner',
            'time': 'created_dt'
        }

        ascend_map = {
            'asc': True,
            'desc': False
        }

        filter_column = BROWSER_WIDGETS['filter_column_dropdown'].get_selected_word()
        filter_ascend = BROWSER_WIDGETS['filter_ascend_dropdown'].get_selected_word()

        self._selected_index = None

        start_row = (self._page_number - 1) * GAMES_PER_PAGE + 1
        end_row = (self._page_number) * GAMES_PER_PAGE
        self._games_list = get_ordered_games(column_map[filter_column], ascend_map[filter_ascend], start_row=start_row, end_row=end_row)

        BROWSER_WIDGETS['browser_strip'].initialise_games_list(self._games_list)
        BROWSER_WIDGETS['browser_strip'].set_surface_size(window.size)
        BROWSER_WIDGETS['scroll_area'].set_image()

    def get_event(self, event):
        widget_event = self._widget_group.process_event(event)

        if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            BROWSER_WIDGETS['help'].kill()

        if widget_event is None:
            return

        match widget_event.type:
            case BrowserEventType.MENU_CLICK:
                self.next = 'menu'
                self.done = True

            case BrowserEventType.BROWSER_STRIP_CLICK:
                self._selected_index = widget_event.selected_index

            case BrowserEventType.COPY_CLICK:
                if self._selected_index is None:
                    return
                logger.info('COPYING TO CLIPBOARD:', self._games_list[self._selected_index]['fen_string'])
                pyperclip.copy(self._games_list[self._selected_index]['fen_string'
    ])

            case BrowserEventType.DELETE_CLICK:
                if self._selected_index is None:
                    return
                delete_game(self._games_list[self._selected_index]['id'])
                self.refresh_games_list()

            case BrowserEventType.REVIEW_CLICK:
                if self._selected_index is None:
                    return

                self.next = 'review'
                self.done = True

            case BrowserEventType.FILTER_COLUMN_CLICK:
                selected_word = BROWSER_WIDGETS['filter_column_dropdown'].
    get_selected_word()

                if selected_word is None:
                    return

                self.refresh_games_list()

            case BrowserEventType.FILTER_ASCEND_CLICK:
                selected_word = BROWSER_WIDGETS['filter_ascend_dropdown'].
    get_selected_word()

                if selected_word is None:
                    return

                self.refresh_games_list()

            case BrowserEventType.PAGE_CLICK:
                self._page_number = widget_event.data

                self.refresh_games_list()

            case BrowserEventType.HELP_CLICK:
                self._widget_group.add(BROWSER_WIDGETS['help'])
                self._widget_group.handle_resize(window.size)

    def draw(self):
        self._widget_group.draw()
```

```
from data.components.custom_event import CustomEvent
from data.constants import BrowserEventType, GAMES_PER_PAGE
from data.assets import GRAPHICS
from data.widgets import *
from data.utils.database_helpers import get_number_of_games

BROWSER_HEIGHT = 0.6

browser_strip = BrowserStrip(
    relative_position=(0.0, 0.0),
    relative_height=BROWSER_HEIGHT,
    games_list=[]
)

number_of_pages = get_number_of_games() // GAMES_PER_PAGE + 1

carousel_widgets = {
    i: Text(
        relative_position=(0, 0),
        relative_size=(0.3, 0.1),
        text=f"PAGE {i} OF {number_of_pages}",
        fill_colour=(0, 0, 0, 0),
        fit_vertical=False,
        border_width=0,
    )
    for i in range(1, number_of_pages + 1)
}

sort_by_container = Rectangle(
    relative_size=(0.5, 0.1),
    relative_position=(0.01, 0.77),
    anchor_x='right',
    visible=True
)

buttons_container = Rectangle(
    relative_position=(0, 0.025),
    relative_size=(0.5, 0.1),
    scale_mode='height',
    anchor_x='center'
)

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.15, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

BROWSER_WIDGETS = {
    'help':
    Icon(
        relative_position=(0, 0),
        relative_size=(1.02, 1.02),
        icon=GRAPHICS['browser_help'],
        anchor_x='center',
        anchor_y='center',
        border_width=0,
        fill_colour=(0, 0, 0, 0)
    ),
    'default': [
        buttons_container,
        sort_by_container,
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            event=CustomEvent(BrowserEventType.MENU_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['help_base'],
            hover_icon=GRAPHICS['help_hover'],
            press_icon=GRAPHICS['help_press'],
            event=CustomEvent(BrowserEventType.HELP_CLICK)
        ),
        ReactiveIconButton(
            parent=buttons_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['copy_base'],
            hover_icon=GRAPHICS['copy_hover'],
            press_icon=GRAPHICS['copy_press'],
            event=CustomEvent(BrowserEventType.COPY_CLICK),
        ),
        ReactiveIconButton(
            parent=buttons_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='center',
            base_icon=GRAPHICS['delete_base'],
            hover_icon=GRAPHICS['delete_hover'],
            press_icon=GRAPHICS['delete_press'],
            event=CustomEvent(BrowserEventType.DELETE_CLICK),
        ),
        ReactiveIconButton(
            parent=buttons_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='right',
            base_icon=GRAPHICS['review_base'],
            hover_icon=GRAPHICS['review_hover'],
            press_icon=GRAPHICS['review_press'],
            event=CustomEvent(BrowserEventType.REVIEW_CLICK),
        ),
        Text(
            parent=sort_by_container,
            relative_position=(0, 0),
            relative_size=(0.3, 1),
            fit_vertical=False,
            text='SORT BY:',
            border_width=0,
            fill_colour=(0, 0, 0, 0)
        )
    ],
    'browser_strip':
        browser_strip,
    'scroll_area':
    ScrollArea(
        relative_position=(0.0, 0.15),
        relative_size=(1, BROWSER_HEIGHT),
        vertical=False,
        widget=browser_strip
    ),
    'filter_column_dropdown':
    Dropdown(
        parent=sort_by_container,
        relative_position=(0.3, 0),
```

```python
        relative_height=0.75,
        anchor_x='right',
        word_list=['time', 'moves', 'winner'],
        fill_colour=(255, 100, 100),
        event=CustomEvent(BrowserEventType.FILTER_COLUMN_CLICK)
    ),
    'filter_ascend_dropdown':
    Dropdown(
        parent=sort_by_container,
        relative_position=(0, 0),
        relative_height=0.75,
        anchor_x='right',
        word_list=['desc', 'asc'],
        fill_colour=(255, 100, 100),
        event=CustomEvent(BrowserEventType.FILTER_ASCEND_CLICK)
    ),
    'page_carousel':
    Carousel(
        relative_position=(0.01, 0.77),
        margin=5,
        widgets_dict=carousel_widgets,
        event=CustomEvent(BrowserEventType.PAGE_CLICK),
    )
}
```

```python
import pygame
from data.constants import ConfigEventType, Colour, ShaderType
from data.states.config.default_config import default_config
from data.states.config.widget_dict import CONFIG_WIDGETS
from data.managers.logs import initialise_logger
from data.managers.animation import animation
from data.managers.window import window
from data.managers.audio import audio
from data.managers.theme import theme
from data.assets import MUSIC, SFX
from data.control import _State
from random import randint

logger = initialise_logger(__name__)

class Config(_State):
    def __init__(self):
        super().__init__()

        self._config = None
        self._valid_fen = True
        self._selected_preset = None

    def cleanup(self):
        super().cleanup()

        window.clear_apply_arguments(ShaderType.BLOOM)

        return self._config

    def startup(self, persist=None):
        super().startup(CONFIG_WIDGETS, music=MUSIC[f'menu_{randint(1, 3)}'])
        window.set_apply_arguments(ShaderType.BLOOM, highlight_colours=[(pygame.Color('0x95e0cc')).rgb, pygame.Color('0xf14e52').rgb], colour_intensity=0.9)

        CONFIG_WIDGETS['invalid_fen_string'].kill()
        CONFIG_WIDGETS['help'].kill()

        self._config = default_config

        if persist:
            self._config['FEN_STRING'] = persist

        self.set_fen_string(self._config['FEN_STRING'])
        self.toggle_pvc(self._config['CPU_ENABLED'])
        self.set_active_colour(self._config['COLOUR'])

        CONFIG_WIDGETS['cpu_depth_carousel'].set_to_key(self._config['CPU_DEPTH'])
        if self._config['CPU_ENABLED']:
            self.create_depth_picker()
        else:
            self.remove_depth_picker()

        self.draw()

    def create_depth_picker(self):
        # CONFIG_WIDGETS['start_button'].update_relative_position((0.5, 0.8))
        # CONFIG_WIDGETS['start_button'].set_image()
        CONFIG_WIDGETS['cpu_depth_carousel'].set_surface_size(window.size)
        CONFIG_WIDGETS['cpu_depth_carousel'].set_image()
        CONFIG_WIDGETS['cpu_depth_carousel'].set_geometry()
        self._widget_group.add(CONFIG_WIDGETS['cpu_depth_carousel'])

    def remove_depth_picker(self):
        # CONFIG_WIDGETS['start_button'].update_relative_position((0.5, 0.7))
        # CONFIG_WIDGETS['start_button'].set_image()

        CONFIG_WIDGETS['cpu_depth_carousel'].kill()

    def toggle_pvc(self, pvc_enabled):
        if pvc_enabled:
            CONFIG_WIDGETS['pvc_button'].set_locked(True)
            CONFIG_WIDGETS['pvp_button'].set_locked(False)
        else:
            CONFIG_WIDGETS['pvp_button'].set_locked(True)
            CONFIG_WIDGETS['pvc_button'].set_locked(False)

        self._config['CPU_ENABLED'] = pvc_enabled

        if self._config['CPU_ENABLED']:
            self.create_depth_picker()
        else:
            self.remove_depth_picker()

    def set_fen_string(self, new_fen_string):
        CONFIG_WIDGETS['fen_string_input'].set_text(new_fen_string)
        self._config['FEN_STRING'] = new_fen_string

        self.set_preset_overlay(new_fen_string)

        try:
            CONFIG_WIDGETS['board_thumbnail'].initialise_board(new_fen_string)
            CONFIG_WIDGETS['invalid_fen_string'].kill()

            if new_fen_string[-1].lower() == 'r':
                self.set_active_colour(Colour.RED)
            else:
                self.set_active_colour(Colour.BLUE)

            self._valid_fen = True
        except:
            CONFIG_WIDGETS['board_thumbnail'].initialise_board('')
            self._widget_group.add(CONFIG_WIDGETS['invalid_fen_string'])

            window.set_effect(ShaderType.SHAKE)
            animation.set_timer(500, lambda: window.clear_effect(ShaderType.SHAKE)
    )

            audio.play_sfx(SFX['error_1'])
            audio.play_sfx(SFX['error_2'])

            self._valid_fen = False

    def get_event(self, event):
        widget_event = self._widget_group.process_event(event)

        if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            CONFIG_WIDGETS['help'].kill()

        if widget_event is None:
            return

        match widget_event.type:
            case ConfigEventType.GAME_CLICK:
                if self._valid_fen:
                    self.next = 'game'
                    self.done = True

            case ConfigEventType.MENU_CLICK:
                self.next = 'menu'
                self.done = True

            case ConfigEventType.TIME_CLICK:
                self._config['TIME_ENABLED'] = not(widget_event.data)
                CONFIG_WIDGETS['timer_button'].set_next_icon()

            case ConfigEventType.PVP_CLICK:
                self.toggle_pvc(False)

            case ConfigEventType.PVC_CLICK:
                self.toggle_pvc(True)

            case ConfigEventType.FEN_STRING_TYPE:
                self.set_fen_string(widget_event.text)

            case ConfigEventType.TIME_TYPE:
                if widget_event.text == '':
                    self._config['TIME'] = 5
                else:
                    self._config['TIME'] = float(widget_event.text)

            case ConfigEventType.CPU_DEPTH_CLICK:
                self._config['CPU_DEPTH'] = int(widget_event.data)

            case ConfigEventType.PRESET_CLICK:
                self.set_fen_string(widget_event.fen_string)

            case ConfigEventType.SETUP_CLICK:
                self.next = 'editor'
                self.done = True

            case ConfigEventType.COLOUR_CLICK:
                self.set_active_colour(widget_event.data.get_flipped_colour())

            case ConfigEventType.HELP_CLICK:
                self._widget_group.add(CONFIG_WIDGETS['help'])
                self._widget_group.handle_resize(window.size)

    def set_preset_overlay(self, fen_string):
        fen_string_widget_map = {
            'sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b': 'preset_1',
            'sc3ncfcncra2/10/3Pd2pa3/paPc2Pbra2pbPd/pbPd2Rapd2paPc/3Pc2pb3/10/2RaNaFaNa3Sa b': 'preset_2',
            'sc3pcncpb3/5fc4/pa3pcncra3/pb1rd1Pd1Pb3/3pd1pb1Rd1Pd/3RaNaPa3Pc/4Fa5/3PdNaPa3Sa b': 'preset_3'
        }

        if fen_string in fen_string_widget_map:
            self._selected_preset = CONFIG_WIDGETS[fen_string_widget_map[fen_string]]
        else:
            self._selected_preset = None

    def set_active_colour(self, colour):
        if self._config['COLOUR'] != colour:
            CONFIG_WIDGETS['to_move_button'].set_next_icon()

        self._config['COLOUR'] = colour

        if colour == Colour.BLUE:
            CONFIG_WIDGETS['to_move_text'].set_text('BLUE TO MOVE')
        elif colour == Colour.RED:
            CONFIG_WIDGETS['to_move_text'].set_text('RED TO MOVE')

        if self._valid_fen:
            self._config['FEN_STRING'] = self._config['FEN_STRING'][:-1] + colour.name[0].lower()
            CONFIG_WIDGETS['fen_string_input'].set_text(self._config['FEN_STRING'])

    def draw(self):
        self._widget_group.draw()

        if self._selected_preset:
            pygame.draw.rect(window.screen, theme['borderPrimary'], (*self._selected_preset.position, *self._selected_preset.size), width=int(theme['borderWidth']))

    def update(self, **kwargs):
        self._widget_group.update()
        super().update(**kwargs)
```

```
from data.constants import Colour

default_config = {
    'CPU_ENABLED': False,
    'CPU_DEPTH': 2,
    'FEN_STRING': 'sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b',
    'TIME_ENABLED': True,
    'TIME': 5,
    'COLOUR': Colour.BLUE,
}
```

```python
import pygame
from data.widgets import *
from data.states.config.default_config import default_config
from data.components.custom_event import CustomEvent
from data.constants import ConfigEventType, Colour
from data.assets import GRAPHICS
from data.utils.asset_helpers import get_highlighted_icon
from data.managers.theme import theme

def float_validator(num_string):
    try:
        float(num_string)
        return True
    except:
        return False

if default_config['CPU_ENABLED']:
    pvp_icons = {False: GRAPHICS['swords'], True: GRAPHICS['swords']}
    pvc_icons = {True: GRAPHICS['robot'], False: GRAPHICS['robot']}
    pvc_locked = True
    pvp_locked = False
else:
    pvp_icons = {True: GRAPHICS['swords'], False: GRAPHICS['swords']}
    pvc_icons = {False: GRAPHICS['robot'], True: GRAPHICS['robot']}
    pvc_locked = False
    pvp_locked = True

if default_config['TIME_ENABLED']:
    time_enabled_icons = {True: GRAPHICS['timer'], False: get_highlighted_icon(GRAPHICS['timer'])}
else:
    time_enabled_icons = {False: get_highlighted_icon(GRAPHICS['timer']), True: GRAPHICS['timer']}

if default_config['COLOUR'] == Colour.BLUE:
    colour_icons = {Colour.BLUE: GRAPHICS['pharoah_0_a'], Colour.RED: GRAPHICS['pharoah_1_a']}
else:
    colour_icons = {Colour.RED: GRAPHICS['pharoah_1_a'], Colour.BLUE: GRAPHICS['pharoah_0_a']}

preview_container = Rectangle(
    relative_position=(-0.15, 0),
    relative_size=(0.65, 0.9),
    anchor_x='center',
    anchor_y='center',
)

config_container = Rectangle(
    relative_position=(0.325, 0),
    relative_size=(0.3, 0.9),
    anchor_x='center',
    anchor_y='center',
)

to_move_container = Rectangle(
    parent=config_container,
    relative_size=(0.9, 0.15),
    relative_position=(0, 0.1),
    anchor_x='center'
```

```
)

board_thumbnail = BoardThumbnail(
    parent=preview_container,
    relative_position=(0, 0),
    relative_width=0.7,
    scale_mode='width',
    anchor_x='right',
)

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.15, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

CONFIG_WIDGETS = {
    'help':
    Icon(
        relative_position=(0, 0),
        relative_size=(1.02, 1.02),
        icon=GRAPHICS['config_help'],
        anchor_x='center',
        anchor_y='center',
        border_width=0,
        fill_colour=(0, 0, 0, 0)
    ),
    'default': [
        preview_container,
        config_container,
        to_move_container,
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            event=CustomEvent(ConfigEventType.MENU_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['help_base'],
            hover_icon=GRAPHICS['help_hover'],
            press_icon=GRAPHICS['help_press'],
            event=CustomEvent(ConfigEventType.HELP_CLICK)
        ),
        TextInput(
            parent=config_container,
            relative_position=(0.3, 0.3),
            relative_size=(0.65, 0.15),
            fit_vertical=True,
            placeholder='TIME CONTROL (DEFAULT 5)',
            default=str(default_config['TIME']),
            border_width=5,
            margin=20,
            validator=float_validator,
            event=CustomEvent(ConfigEventType.TIME_TYPE)
        ),
        Text(
            parent=config_container,
            fit_vertical=False,
            relative_position=(0.75, 0.3),
            relative_size=(0.2, 0.15),
            text='MINS',
            border_width=0,
            fill_colour=(0, 0, 0, 0)
        ),
        TextButton(
            parent=preview_container,
            relative_position=(0.3, 0),
            relative_size=(0.15, 0.15),
            text='CUSTOM',
            anchor_y='bottom',
            fit_vertical=False,
            margin=10,
            event=CustomEvent(ConfigEventType.SETUP_CLICK)
        )
    ],
    'board_thumbnail':
        board_thumbnail,
    'fen_string_input':
    TextInput(
        parent=preview_container,
        relative_position=(0, 0),
        relative_size=(0.55, 0.15),
        fit_vertical=False,
        placeholder='ENTER FEN STRING',
        default='sc3ncfcncpb2/2pc7/3Pd7/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b',
        border_width=5,
        anchor_y='bottom',
        anchor_x='right',
        margin=20,
        event=CustomEvent(ConfigEventType.FEN_STRING_TYPE)
    ),
    'start_button':
    TextButton(
        parent=config_container,
        relative_position=(0, 0),
        relative_size=(0.9, 0.3),
        anchor_y='bottom',
        anchor_x='center',
        text='START NEW GAME',
        strength=0.1,
        text_colour=theme['textSecondary'],
        margin=20,
        fit_vertical=False,
        event=CustomEvent(ConfigEventType.GAME_CLICK)
    ),
    'timer_button':
    MultipleIconButton(
        parent=config_container,
        scale_mode='height',
        relative_position=(0.05, 0.3),
        relative_size=(0.15, 0.15),
        margin=10,
        border_width=5,
        border_radius=5,
        icons_dict=time_enabled_icons,
        event=CustomEvent(ConfigEventType.TIME_CLICK)
    ),
    'pvp_button':
    MultipleIconButton(
        parent=config_container,
        relative_position=(-0.225, 0.5),
        relative_size=(0.45, 0.15),
        margin=15,
        anchor_x='center',
        icons_dict=pvp_icons,
        stretch=False,
        event=CustomEvent(ConfigEventType.PVP_CLICK)
    ),
    'pvc_button':
    MultipleIconButton(
        parent=config_container,
        relative_position=(0.225, 0.5),
        relative_size=(0.45, 0.15),
        anchor_x='center',
        margin=15,
        icons_dict=pvc_icons,
        stretch=False,
        event=CustomEvent(ConfigEventType.PVC_CLICK)
    ),
    'invalid_fen_string':
    Text(
        parent=board_thumbnail,
        relative_position=(0, 0),
        relative_size=(0.9, 0.1),
        fit_vertical=False,
        anchor_x='center',
        anchor_y='center',
        text='INVALID FEN STRING!',
        margin=10,
        fill_colour=theme['fillError'],
        text_colour=theme['textError'],
    ),
    'preset_1':
    BoardThumbnailButton(
        parent=preview_container,
        relative_width=0.25,
        relative_position=(0, 0),
        scale_mode='width',
        fen_string="sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b",
        event=CustomEvent(ConfigEventType.PRESET_CLICK)
    ),
    'preset_2':
    BoardThumbnailButton(
        parent=preview_container,
        relative_width=0.25,
        relative_position=(0, 0.35),
        scale_mode='width',
        fen_string="sc3ncfcncra2/10/3Pd2pa3/paPc2Pbra2pbPd/pbPd2Rapd2paPc/3Pc2pb3/10/2RaNaFaNa3Sa b",
        event=CustomEvent(ConfigEventType.PRESET_CLICK)
    ),
    'preset_3':
```

```
BoardThumbnailButton(
    parent=preview_container,
    relative_width=0.25,
    relative_position=(0, 0.7),
    scale_mode='width',
    fen_string="sc3pcncpb3/5fc4/pa3pcncra3/pb1rd1Pd1Pb3/3pd1pb1Rd1Pd/3RaNaPa3Pc/4Fa5/3PdNaPa3Sa b",
    event=CustomEvent(ConfigEventType.PRESET_CLICK)
),
'to_move_button':
MultipleIconButton(
    parent=to_move_container,
    scale_mode='height',
    relative_position=(0, 0),
    relative_size=(1, 1),
    icons_dict=colour_icons,
    anchor_x='left',
    event=CustomEvent(ConfigEventType.COLOUR_CLICK)
),
'to_move_text':
Text(
    parent=to_move_container,
    relative_position=(0, 0),
    relative_size=(0.75, 1),
    fit_vertical=False,
    text='TO MOVE',
    anchor_x='right'
),
'cpu_depth_carousel':
Carousel(
    parent=config_container,
    relative_position=(0, 0.65),
    event=CustomEvent(ConfigEventType.CPU_DEPTH_CLICK),
    anchor_x='center',
    border_width=0,
    fill_colour=(0, 0, 0, 0),
    widgets_dict={
        2: Text(
            parent=config_container,
            relative_position=(0, 0),
            relative_size=(0.8, 0.075),
            text="EASY",
            margin=0,
            border_width=0,
            fill_colour=(0, 0, 0, 0)
        ),
        3: Text(
            parent=config_container,
            relative_position=(0, 0),
            relative_size=(0.8, 0.075),
            text="MEDIUM",
            margin=0,
            border_width=0,
            fill_colour=(0, 0, 0, 0)
        ),
        4: Text(
            parent=config_container,
            relative_position=(0, 0),
            relative_size=(0.8, 0.075),
            text="HARD",
            margin=0,
            border_width=0,
                fill_colour=(0, 0, 0, 0)
            ),
        }
    )
}
```

```python
import pygame
import pyperclip
from data.constants import EditorEventType, Colour, RotationDirection, Piece, Rotation
from data.states.game.components.bitboard_collection import BitboardCollection
from data.states.game.components.fen_parser import encode_fen_string
from data.states.game.components.overlay_draw import OverlayDraw
from data.states.game.components.piece_group import PieceGroup
from data.states.game.components.father import DragAndDrop
from data.utils.bitboard_helpers import coords_to_bitboard
from data.states.editor.widget_dict import EDITOR_WIDGETS
from data.utils.board_helpers import screen_pos_to_coords
from data.managers.logs import initialise_logger
from data.managers.window import window
from data.control import _State

logger = initialise_logger(__name__)

class Editor(_State):
    def __init__(self):
        super().__init__()

        self._bitboards = None
        self._piece_group = None
        self._selected_coords = None
        self._selected_tool = None
        self._selected_tool_colour = None
        self._initial_fen_string = None
        self._starting_colour = None

        self._drag_and_drop = None
        self._overlay_draw = None

    def cleanup(self):
        super().cleanup()

        self.deselect_tool()

        return encode_fen_string(self._bitboards)

    def startup(self, persist):
        super().startup(EDITOR_WIDGETS)
        EDITOR_WIDGETS['help'].kill()

        self._drag_and_drop = DragAndDrop(EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)
        self._overlay_draw = OverlayDraw(EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)
        self._bitboards = BitboardCollection(persist['FEN_STRING'])
        self._piece_group = PieceGroup()

        self._selected_coords = None
        self._selected_tool = None
        self._selected_tool_colour = None
        self._initial_fen_string = persist['FEN_STRING']
        self._starting_colour = Colour.BLUE

        self.refresh_pieces()
        self.set_starting_colour(Colour.BLUE if persist['FEN_STRING'][-1].lower() == 'b' else Colour.RED)
        self.draw()

    @property
    def selected_coords(self):
        return self._selected_coords

    @selected_coords.setter
    def selected_coords(self, new_coords):
        self._overlay_draw.set_selected_coords(new_coords)
        self._selected_coords = new_coords

    def get_event(self, event):
        widget_event = self._widget_group.process_event(event)

        if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            EDITOR_WIDGETS['help'].kill()

        if event.type == pygame.MOUSEBUTTONDOWN:
            clicked_coords = screen_pos_to_coords(event.pos, EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)

            if clicked_coords:
                self.selected_coords = clicked_coords

                if self._selected_tool is None:
                    return

                if self._selected_tool == 'MOVE':
                    self.set_dragged_piece(clicked_coords)

                elif self._selected_tool == 'ERASE':
                    self.remove_piece()
                else:
                    self.set_piece(self._selected_tool, self._selected_tool_colour, Rotation.UP)

                return

        if event.type == pygame.MOUSEBUTTONUP:
            clicked_coords = screen_pos_to_coords(event.pos, EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)

            if self._drag_and_drop.dragged_sprite:
                self.remove_dragged_piece(clicked_coords)
                return

        if widget_event is None:
            if event.type == pygame.MOUSEBUTTONDOWN and self._widget_group.on_widget(event.pos) is False:
                self.selected_coords = None

            return

        match widget_event.type:
            case None:
                return

            case EditorEventType.MENU_CLICK:
```

```
            self.next = 'menu'
            self.done = True

        case EditorEventType.PICK_PIECE_CLICK:
            if widget_event.piece == self._selected_tool and widget_event.active_colour == self._selected_tool_colour:
                self.deselect_tool()
            else:
                self.select_tool(widget_event.piece, widget_event.active_colour)

        case EditorEventType.ROTATE_PIECE_CLICK:
            self.rotate_piece(widget_event.rotation_direction)

        case EditorEventType.EMPTY_CLICK:
            self._bitboards = BitboardCollection(fen_string='sc9/10/10/10/10/10/10/9Sa b')
            self.refresh_pieces()

        case EditorEventType.RESET_CLICK:
            self.reset_board()

        case EditorEventType.COPY_CLICK:
            logger.info(f'COPYING TO CLIPBOARD: {encode_fen_string(self._bitboards)}')
            pyperclip.copy(encode_fen_string(self._bitboards))

        case EditorEventType.BLUE_START_CLICK:
            self.set_starting_colour(Colour.BLUE)

        case EditorEventType.RED_START_CLICK:
            self.set_starting_colour(Colour.RED)

        case EditorEventType.START_CLICK:
            self.next = 'config'
            self.done = True

        case EditorEventType.CONFIG_CLICK:
            self.reset_board()
            self.next = 'config'
            self.done = True

        case EditorEventType.ERASE_CLICK:
            if self._selected_tool == 'ERASE':
                self.deselect_tool()
            else:
                self.select_tool('ERASE', None)

        case EditorEventType.MOVE_CLICK:
            if self._selected_tool == 'MOVE':
                self.deselect_tool()
            else:
                self.select_tool('MOVE', None)

        case EditorEventType.HELP_CLICK:
            self._widget_group.add(EDITOR_WIDGETS['help'])
            self._widget_group.handle_resize(window.size)

def reset_board(self):
    self._bitboards = BitboardCollection(self._initial_fen_string)
    self.refresh_pieces()

```

```python
    def refresh_pieces(self):
        self._piece_group.initialise_pieces(self._bitboards.convert_to_piece_list(), EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)

    def set_starting_colour(self, new_colour):
        if new_colour == Colour.BLUE:
            EDITOR_WIDGETS['blue_start_button'].set_locked(True)
            EDITOR_WIDGETS['red_start_button'].set_locked(False)
        elif new_colour == Colour.RED:
            EDITOR_WIDGETS['blue_start_button'].set_locked(False)
            EDITOR_WIDGETS['red_start_button'].set_locked(True)

        if new_colour != self._starting_colour:
            EDITOR_WIDGETS['blue_start_button'].set_next_icon()
            EDITOR_WIDGETS['red_start_button'].set_next_icon()

        self._starting_colour = new_colour
        self._bitboards.active_colour = new_colour

    def set_dragged_piece(self, coords):
        bitboard_under_mouse = coords_to_bitboard(coords)
        dragged_piece = self._bitboards.get_piece_on(bitboard_under_mouse, Colour.BLUE) or self._bitboards.get_piece_on(bitboard_under_mouse, Colour.RED)

        if dragged_piece is None:
            return

        dragged_colour = self._bitboards.get_colour_on(bitboard_under_mouse)
        dragged_rotation = self._bitboards.get_rotation_on(bitboard_under_mouse)

        self._drag_and_drop.set_dragged_piece(dragged_piece, dragged_colour, dragged_rotation)
        self._overlay_draw.set_hover_limit(False)

    def remove_dragged_piece(self, coords):
        piece, colour, rotation = self._drag_and_drop.get_dragged_info()

        if coords and coords != self._selected_coords and piece != Piece.SPHINX:
            self.remove_piece()
            self.selected_coords = coords
            self.set_piece(piece, colour, rotation)
            self.selected_coords = None

        self._drag_and_drop.remove_dragged_piece()
        self._overlay_draw.set_hover_limit(True)

    def set_piece(self, piece, colour, rotation):
        if self.selected_coords is None or self.selected_coords == (0, 7) or self.selected_coords == (9, 0):
            return

        self.remove_piece()

        selected_bitboard = coords_to_bitboard(self.selected_coords)
        self._bitboards.set_square(selected_bitboard, piece, colour)
        self._bitboards.set_rotation(selected_bitboard, rotation)

        self.refresh_pieces()

    def remove_piece(self):
        if self.selected_coords is None or self.selected_coords == (0, 7) or self.selected_coords == (9, 0):
            return

        selected_bitboard = coords_to_bitboard(self.selected_coords)
        self._bitboards.clear_square(selected_bitboard, Colour.BLUE)
        self._bitboards.clear_square(selected_bitboard, Colour.RED)
        self._bitboards.clear_rotation(selected_bitboard)

        self.refresh_pieces()

    def rotate_piece(self, rotation_direction):
        if self.selected_coords is None or self.selected_coords == (0, 7) or self.selected_coords == (9, 0):
            return

        selected_bitboard = coords_to_bitboard(self.selected_coords)

        if self._bitboards.get_piece_on(selected_bitboard, Colour.BLUE) is None and self._bitboards.get_piece_on(selected_bitboard, Colour.RED) is None:
            return

        current_rotation = self._bitboards.get_rotation_on(selected_bitboard)

        if rotation_direction == RotationDirection.CLOCKWISE:
            self._bitboards.update_rotation(selected_bitboard, selected_bitboard, current_rotation.get_clockwise())
        elif rotation_direction == RotationDirection.ANTICLOCKWISE:
            self._bitboards.update_rotation(selected_bitboard, selected_bitboard, current_rotation.get_anticlockwise())

        self.refresh_pieces()

    def select_tool(self, piece, colour):
        dict_name_map = { Colour.BLUE: 'blue_piece_buttons', Colour.RED: 'red_piece_buttons' }

        self.deselect_tool()

        if piece == 'ERASE':
            EDITOR_WIDGETS['erase_button'].set_locked(True)
            EDITOR_WIDGETS['erase_button'].set_next_icon()
        elif piece == 'MOVE':
            EDITOR_WIDGETS['move_button'].set_locked(True)
            EDITOR_WIDGETS['move_button'].set_next_icon()
        else:
            EDITOR_WIDGETS[dict_name_map[colour]][piece].set_locked(True)
            EDITOR_WIDGETS[dict_name_map[colour]][piece].set_next_icon()

        self._selected_tool = piece
        self._selected_tool_colour = colour

    def deselect_tool(self):
        dict_name_map = { Colour.BLUE: 'blue_piece_buttons', Colour.RED: 'red_piece_buttons' }

        if self._selected_tool:
            if self._selected_tool == 'ERASE':
                EDITOR_WIDGETS['erase_button'].set_locked(False)
                EDITOR_WIDGETS['erase_button'].set_next_icon()
            elif self._selected_tool == 'MOVE':
                EDITOR_WIDGETS['move_button'].set_locked(False)
                EDITOR_WIDGETS['move_button'].set_next_icon()
            else:
                EDITOR_WIDGETS[dict_name_map[self._selected_tool_colour]][self._selected_tool].set_locked(False)
                EDITOR_WIDGETS[dict_name_map[self._selected_tool_colour]][self._selected_tool].set_next_icon()

        self._selected_tool = None
        self._selected_tool_colour = None

    def handle_resize(self):
        super().handle_resize()
        self._piece_group.handle_resize(EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)
        self._drag_and_drop.handle_resize(EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)
        self._overlay_draw.handle_resize(EDITOR_WIDGETS['chessboard'].position, EDITOR_WIDGETS['chessboard'].size)

    def draw(self):
        self._widget_group.draw()
        self._overlay_draw.draw(window.screen)
        self._piece_group.draw(window.screen)
        self._drag_and_drop.draw(window.screen)
```

```
from data.constants import Piece, Colour, RotationDirection, EditorEventType, BLUE_BUTTON_COLOURS
from data.utils.asset_helpers import get_highlighted_icon
from data.components.custom_event import CustomEvent
from data.assets import GRAPHICS
from data.widgets import *

blue_pieces_container = Rectangle(
    relative_position=(0.25, 0),
    relative_size=(0.13, 0.65),
    scale_mode='height',
    anchor_y='center',
    anchor_x='center'
)

red_pieces_container = Rectangle(
    relative_position=(-0.25, 0),
    relative_size=(0.13, 0.65),
    scale_mode='height',
    anchor_y='center',
    anchor_x='center'
)

bottom_actions_container = Rectangle(
    relative_position=(0, 0.05),
    relative_size=(0.4, 0.1),
    anchor_x='center',
    anchor_y='bottom'
)

top_actions_container = Rectangle(
    relative_position=(0, 0.05),
    relative_size=(0.3, 0.1),
    anchor_x='center',
    scale_mode='height'
)

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.15, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

EDITOR_WIDGETS = {
    'help':
    Icon(
        relative_position=(0, 0),
        relative_size=(1.02, 1.02),
        icon=GRAPHICS['editor_help'],
        anchor_x='center',
        anchor_y='center',
        border_width=0,
        fill_colour=(0, 0, 0, 0)
    ),
    'default': [
        red_pieces_container,
        blue_pieces_container,
        bottom_actions_container,
        top_actions_container,
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            event=CustomEvent(EditorEventType.MENU_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['help_base'],
            hover_icon=GRAPHICS['help_hover'],
            press_icon=GRAPHICS['help_press'],
            event=CustomEvent(EditorEventType.HELP_CLICK)
        ),
        ReactiveIconButton(
            parent=bottom_actions_container,
            relative_position=(0.06, 0),
            relative_size=(1, 1),
            anchor_x='center',
            scale_mode='height',
            base_icon=GRAPHICS['clockwise_arrow_base'],
            hover_icon=GRAPHICS['clockwise_arrow_hover'],
            press_icon=GRAPHICS['clockwise_arrow_press'],
            event=CustomEvent(EditorEventType.ROTATE_PIECE_CLICK, rotation_direction=RotationDirection.CLOCKWISE)
        ),
        ReactiveIconButton(
            parent=bottom_actions_container,
            relative_position=(-0.06, 0),
            relative_size=(1, 1),
            anchor_x='center',
            scale_mode='height',
            base_icon=GRAPHICS['anticlockwise_arrow_base'],
            hover_icon=GRAPHICS['anticlockwise_arrow_hover'],
            press_icon=GRAPHICS['anticlockwise_arrow_press'],
            event=CustomEvent(EditorEventType.ROTATE_PIECE_CLICK,
    rotation_direction=RotationDirection.ANTICLOCKWISE)
        ),
        ReactiveIconButton(
            parent=top_actions_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='right',
            base_icon=GRAPHICS['copy_base'],
            hover_icon=GRAPHICS['copy_hover'],
            press_icon=GRAPHICS['copy_press'],
            event=CustomEvent(EditorEventType.COPY_CLICK),
        ),
        ReactiveIconButton(
            parent=top_actions_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['delete_base'],
            hover_icon=GRAPHICS['delete_hover'],
            press_icon=GRAPHICS['delete_press'],
            event=CustomEvent(EditorEventType.EMPTY_CLICK),
        ),
        ReactiveIconButton(
            parent=top_actions_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='center',
            base_icon=GRAPHICS['discard_arrow_base'],
            hover_icon=GRAPHICS['discard_arrow_hover'],
            press_icon=GRAPHICS['discard_arrow_press'],
            event=CustomEvent(EditorEventType.RESET_CLICK),
        ),
        ReactiveIconButton(
            relative_position=(0, 0),
            fixed_position=(10, 0),
            relative_size=(0.1, 0.1),
            anchor_x='right',
            anchor_y='center',
            scale_mode='height',
            base_icon=GRAPHICS['play_arrow_base'],
            hover_icon=GRAPHICS['play_arrow_hover'],
            press_icon=GRAPHICS['play_arrow_press'],
            event=CustomEvent(EditorEventType.START_CLICK),
        ),
        ReactiveIconButton(
            relative_position=(0, 0),
            fixed_position=(10, 0),
            relative_size=(0.1, 0.1),
            anchor_y='center',
            scale_mode='height',
            base_icon=GRAPHICS['return_arrow_base'],
            hover_icon=GRAPHICS['return_arrow_hover'],
            press_icon=GRAPHICS['return_arrow_press'],
            event=CustomEvent(EditorEventType.CONFIG_CLICK),
        )
    ],
```

```python
    'blue_piece_buttons': {},
    'red_piece_buttons': {},
    'erase_button':
    MultipleIconButton(
        parent=red_pieces_container,
        relative_position=(0, 0),
        relative_size=(0.2, 0.2),
        scale_mode='height',
        margin=10,
        icons_dict={True: GRAPHICS['eraser'], False: get_highlighted_icon(GRAPHICS['eraser'])},
        event=CustomEvent(EditorEventType.ERASE_CLICK),
    ),
    'move_button':
    MultipleIconButton(
        parent=blue_pieces_container,
        relative_position=(0, 0),
        relative_size=(0.2, 0.2),
        scale_mode='height',
        box_colours=BLUE_BUTTON_COLOURS,
        icons_dict={True: GRAPHICS['finger'], False: get_highlighted_icon(GRAPHICS['finger'])},
        event=CustomEvent(EditorEventType.MOVE_CLICK),
    ),
    'chessboard':
    Chessboard(
        relative_position=(0, 0),
        relative_width=0.4,
        scale_mode='width',
        anchor_x='center',
        anchor_y='center'
    ),
    'blue_start_button':
    MultipleIconButton(
        parent=bottom_actions_container,
        relative_position=(0, 0),
        relative_size=(1, 1),
        scale_mode='height',
        anchor_x='right',
        box_colours=BLUE_BUTTON_COLOURS,
        icons_dict={False: get_highlighted_icon(GRAPHICS['pharoah_0_a']), True: GRAPHICS['pharoah_0_a']},
        event=CustomEvent(EditorEventType.BLUE_START_CLICK)
    ),
    'red_start_button':
    MultipleIconButton(
        parent=bottom_actions_container,
        relative_position=(0, 0),
        relative_size=(1, 1),
        scale_mode='height',
        icons_dict={True: GRAPHICS['pharoah_1_a'], False: get_highlighted_icon(GRAPHICS['pharoah_1_a'])},
        event=CustomEvent(EditorEventType.RED_START_CLICK)
    )
}

for index, piece in enumerate([piece for piece in Piece if piece != Piece.SPHINX]):
    blue_icon = GRAPHICS[f'{piece.name.lower()}_0_a']
    dimmed_blue_icon = get_highlighted_icon(blue_icon)

    EDITOR_WIDGETS['blue_piece_buttons'][piece] = MultipleIconButton(
        parent=blue_pieces_container,
        relative_position=(0, (index + 1) / 5),
        relative_size=(0.2, 0.2),
        scale_mode='height',
        box_colours=BLUE_BUTTON_COLOURS,
        icons_dict={True: blue_icon, False: dimmed_blue_icon},
        event=CustomEvent(EditorEventType.PICK_PIECE_CLICK, piece=piece, active_colour=Colour.BLUE)
    )

    red_icon = GRAPHICS[f'{piece.name.lower()}_1_a']

    dimmed_red_icon = get_highlighted_icon(red_icon)

    EDITOR_WIDGETS['red_piece_buttons'][piece] = MultipleIconButton(
        parent=red_pieces_container,
        relative_position=(0, (index + 1) / 5),
        relative_size=(0.2, 0.2),
        scale_mode='height',
        icons_dict={True: red_icon, False: dimmed_red_icon},
        event=CustomEvent(EditorEventType.PICK_PIECE_CLICK, piece=piece, active_colour=Colour.RED)
    )
```

```
import pygame
from functools import partial
from data.states.game.mvc.game_controller import GameController
from data.utils.database_helpers import insert_into_games
from data.states.game.mvc.game_model import GameModel
from data.states.game.mvc.pause_view import PauseView
from data.states.game.mvc.game_view import GameView
from data.states.game.mvc.win_view import WinView
from data.components.game_entry import GameEntry
from data.managers.logs import initialise_logger
from data.managers.window import window
from data.managers.audio import audio
from data.constants import ShaderType
from data.assets import MUSIC, SFX
from data.control import _State

logger = initialise_logger(__name__)

class Game(_State):
    def __init__(self):
        super().__init__()

    def cleanup(self):
        super().cleanup()

        window.clear_apply_arguments(ShaderType.BLOOM)
        window.clear_effect(ShaderType.RAYS)

        game_entry = GameEntry(self.model.states, final_fen_string=self.model.get_fen_string())
        inserted_game = insert_into_games(game_entry.convert_to_row())

        return inserted_game

    def switch_to_menu(self):
        self.next = 'menu'
        self.done = True

```

```python
    def switch_to_review(self):
        self.next = 'review'
        self.done = True

    def startup(self, persist):
        music = MUSIC[['cpu_easy', 'cpu_medium', 'cpu_hard'][persist['CPU_DEPTH'] - 2]] if persist['CPU_ENABLED'] else MUSIC['pvp']
        super().startup(music=music)

        window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.BACKGROUND_LASERS)
        window.set_apply_arguments(ShaderType.BLOOM, highlight_colours=[(pygame.Color('0x95e0cc')).rgb, pygame.Color('0xf14e52').rgb], colour_intensity=0.8)
        binded_startup = partial(self.startup, persist)

        self.model = GameModel(persist)
        self.view = GameView(self.model)
        self.pause_view = PauseView(self.model)
        self.win_view = WinView(self.model)
        self.controller = GameController(self.model, self.view, self.win_view, self.pause_view, self.switch_to_menu, self.switch_to_review, binded_startup)

        self.view.draw()

        audio.play_sfx(SFX['game_start_1'])
        audio.play_sfx(SFX['game_start_2'])

    def get_event(self, event):
        self.controller.handle_event(event)

    def handle_resize(self):
        self.view.handle_resize()
        self.win_view.handle_resize()
        self.pause_view.handle_resize()

    def draw(self):
        self.view.draw()
        self.win_view.draw()
        self.pause_view.draw()

    def update(self):
        self.controller.check_cpu()
        super().update()
```

```python
from data.widgets import *
from data.components.custom_event import CustomEvent
from data.constants import GameEventType, RotationDirection, Colour
from data.assets import GRAPHICS

right_container = Rectangle(
    relative_position=(0.05, 0),
    relative_size=(0.2, 0.5),
    anchor_y='center',
    anchor_x='right',
)

rotate_container = Rectangle(
    relative_position=(0, 0.05),
    relative_size=(0.2, 0.1),
    anchor_x='center',
    anchor_y='bottom',
)
```

```

move_list = MoveList(
    parent=right_container,
    relative_position=(0, 0),
    relative_width=1,
    minimum_height=300,
    move_list=[]
)

resign_button = TextButton(
    parent=right_container,
    relative_position=(0, 0),
    relative_size=(0.5, 0.2),
    fit_vertical=False,
    anchor_y='bottom',
    text="   Resign",
    margin=5,
    event=CustomEvent(GameEventType.RESIGN_CLICK)
)

draw_button = TextButton(
    parent=right_container,
    relative_position=(0, 0),
    relative_size=(0.5, 0.2),
    fit_vertical=False,
    anchor_x='right',
    anchor_y='bottom',
    text="   Draw",
    margin=5,
    event=CustomEvent(GameEventType.DRAW_CLICK)
)

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.225, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

GAME_WIDGETS = {
    'help':
    Icon(
        relative_position=(0, 0),
        relative_size=(1.02, 1.02),
        icon=GRAPHICS['game_help'],
        anchor_x='center',
        anchor_y='center',
        border_width=0,
        fill_colour=(0, 0, 0, 0)
    ),
    'tutorial':
    Icon(
        relative_position=(0, 0),
        relative_size=(0.9, 0.9),
        icon=GRAPHICS['game_tutorial'],
        anchor_x='center',
        anchor_y='center',
    ),
    'default': [
        right_container,
        rotate_container,
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            event=CustomEvent(GameEventType.MENU_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['tutorial_base'],
            hover_icon=GRAPHICS['tutorial_hover'],
            press_icon=GRAPHICS['tutorial_press'],
            event=CustomEvent(GameEventType.TUTORIAL_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0.33, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['help_base'],
            hover_icon=GRAPHICS['help_hover'],
            press_icon=GRAPHICS['help_press'],
            event=CustomEvent(GameEventType.HELP_CLICK)
        ),
        ReactiveIconButton(
            parent=rotate_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='right',
            base_icon=GRAPHICS['clockwise_arrow_base'],
            hover_icon=GRAPHICS['clockwise_arrow_hover'],
            press_icon=GRAPHICS['clockwise_arrow_press'],
            event=CustomEvent(GameEventType.ROTATE_PIECE, rotation_direction=RotationDirection.CLOCKWISE)
        ),
        ReactiveIconButton(
            parent=rotate_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['anticlockwise_arrow_base'],
            hover_icon=GRAPHICS['anticlockwise_arrow_hover'],
            press_icon=GRAPHICS['anticlockwise_arrow_press'],
            event=CustomEvent(GameEventType.ROTATE_PIECE, rotation_direction=RotationDirection.ANTICLOCKWISE)
        ),
        resign_button,
        draw_button,
        Icon(
            parent=resign_button,
            relative_position=(0, 0),
            relative_size=(0.75, 0.75),
            fill_colour=(0, 0, 0, 0),
            scale_mode='height',
            anchor_y='center',
            border_radius=0,
            border_width=0,
            margin=5,
            icon=GRAPHICS['resign']
        ),
        Icon(
            parent=draw_button,
            relative_position=(0, 0),
            relative_size=(0.75, 0.75),
            fill_colour=(0, 0, 0, 0),
            scale_mode='height',
            anchor_y='center',
            border_radius=0,
            border_width=0,
            margin=5,
            icon=GRAPHICS['draw']
        ),
    ],
    'scroll_area': # REMEMBER SCROLL AREA AFTER CONTAINER FOR RESIZING
    ScrollArea(
        parent=right_container,
        relative_position=(0, 0),
        relative_size=(1, 0.8),
        vertical=True,
        widget=move_list
    ),
    'move_list':
        move_list,
    'blue_timer':
    Timer(
        relative_position=(0.05, 0.05),
        anchor_y='center',
        relative_size=(0.1, 0.1),
        active_colour=Colour.BLUE,
        event=CustomEvent(GameEventType.TIMER_END),
    ),
    'red_timer':
    Timer(
        relative_position=(0.05, -0.05),
        anchor_y='center',
        relative_size=(0.1, 0.1),
        active_colour=Colour.RED,
        event=CustomEvent(GameEventType.TIMER_END),
    ),
    'status_text':
    Text(
        relative_position=(0, 0.05),
        relative_size=(0.4, 0.1),
        anchor_x='center',
        fit_vertical=False,
        margin=10,
        text="g",
        minimum_width=400
    ),
    'chessboard':
    Chessboard(
        relative_position=(0, 0),
        anchor_x='center',
        anchor_y='center',
        scale_mode='width',
        relative_width=0.4
    ),
    'blue_piece_display':
    PieceDisplay(
        relative_position=(0.05, 0.05),
        relative_size=(0.2, 0.1),
        anchor_y='bottom',
        active_colour=Colour.BLUE
    ),
    'red_piece_display':
    PieceDisplay(
        relative_position=(0.05, 0.05),
        relative_size=(0.2, 0.1),
        active_colour=Colour.RED
    )
}

PAUSE_WIDGETS = {
    'default': [
        TextButton(
            relative_position=(0, -0.125),
            relative_size=(0.3, 0.2),
            anchor_x='center',
            anchor_y='center',
            text='GO TO MENU',
            fit_vertical=False,
            event=CustomEvent(GameEventType.MENU_CLICK)
        ),
        TextButton(
            relative_position=(0, 0.125),
            relative_size=(0.3, 0.2),
            anchor_x='center',
            anchor_y='center',
            text='RESUME GAME',
            fit_vertical=False,
            event=CustomEvent(GameEventType.PAUSE_CLICK)
        )
    ]
}

win_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.4, 0.8),
    scale_mode='height',
    anchor_x='center',
    anchor_y='center',
    fill_colour=(128, 128, 128, 200),
    visible=True
)

WIN_WIDGETS = {
    'default': [
        win_container,
        TextButton(
            parent=win_container,
            relative_position=(0, 0.5),
            relative_size=(0.8, 0.15),
            text='GO TO MENU',
            anchor_x='center',
            fit_vertical=False,
            event=CustomEvent(GameEventType.MENU_CLICK)
        ),
        TextButton(
            parent=win_container,
            relative_position=(0, 0.65),
            relative_size=(0.8, 0.15),
            text='REVIEW GAME',
            anchor_x='center',
            fit_vertical=False,
            event=CustomEvent(GameEventType.REVIEW_CLICK)
        ),
        TextButton(
            parent=win_container,
            relative_position=(0, 0.8),
            relative_size=(0.8, 0.15),
            text='NEW GAME',
            anchor_x='center',
            fit_vertical=False,
            event=CustomEvent(GameEventType.GAME_CLICK)
        ),
    ],
    'blue_won':
    Icon(
        parent=win_container,
        relative_position=(0, 0.05),
        relative_size=(0.8, 0.3),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['blue_won'],
        fill_colour=(0, 0, 0, 0),
    ),
    'red_won':
    Icon(
        parent=win_container,
        relative_position=(0, 0.05),
        relative_size=(0.8, 0.3),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['red_won'],
        fill_colour=(0, 0, 0, 0),
        fit_icon=True,
    ),
    'draw_won':
    Icon(
        parent=win_container,
        relative_position=(0, 0.05),
        relative_size=(0.8, 0.3),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['draw_won'],
        fill_colour=(0, 0, 0, 0),
    ),
    'by_checkmate':
    Icon(
        parent=win_container,
        relative_position=(0, 0.375),
        relative_size=(0.8, 0.1),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['by_checkmate'],
        fill_colour=(0, 0, 0, 0),
    ),
    'by_resignation':
    Icon(
        parent=win_container,
        relative_position=(0, 0.375),
        relative_size=(0.8, 0.1),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['by_resignation'],
        fill_colour=(0, 0, 0, 0),
    ),
    'by_draw':
    Icon(
        parent=win_container,
        relative_position=(0, 0.375),
        relative_size=(0.8, 0.1),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['by_draw'],
        fill_colour=(0, 0, 0, 0),
    ),
    'by_timeout':
    Icon(
        parent=win_container,
        relative_position=(0, 0.375),
        relative_size=(0.8, 0.1),
        anchor_x='center',
        border_width=0,
        margin=0,
        icon=GRAPHICS['by_timeout'],
        fill_colour=(0, 0, 0, 0),
    )
}
```

```
from data.constants import Rank, File, Piece, Colour, Rotation, RotationIndex, EMPTY_BB
from data.states.game.components.fen_parser import parse_fen_string
from data.states.game.cpu.zobrist_hasher import ZobristHasher
from data.utils import bitboard_helpers as bb_helpers
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class BitboardCollection:
    def __init__(self, fen_string):
        self.piece_bitboards = [{char: EMPTY_BB for char in Piece}, {char: EMPTY_BB for char in Piece}]
        self.combined_colour_bitboards = [EMPTY_BB, EMPTY_BB]
        self.combined_all_bitboard = EMPTY_BB
        self.rotation_bitboards = [EMPTY_BB, EMPTY_BB]
        self.active_colour = Colour.BLUE
        self._hasher = ZobristHasher()

        try:
            if fen_string:
                self.piece_bitboards, self.combined_colour_bitboards, self.combined_all_bitboard, self.rotation_bitboards, self.active_colour = parse_fen_string(fen_string)
                self.initialise_hash()
        except ValueError as error:
            logger.error('Please input a valid FEN string:', error)
            raise error

    def __str__(self):
        """
        Returns a string representation of the bitboards.

        Returns:
            str: Bitboards formatted with piece type and colour shown.
        """
        characters = ''
        for rank in reversed(Rank):
            for file in File:
                bitboard = 1 << (rank * 10 + file)

                colour = self.get_colour_on(bitboard)
                piece = self.get_piece_on(bitboard, Colour.BLUE) or self.get_piece_on(bitboard, Colour.RED)

                if piece is not None:
                        characters += f'{piece.upper() if colour == Colour.BLUE else piece}  '
                else:
                    characters += '.  '

            characters += '\n\n'

        return characters

    def get_rotation_string(self):
        """
        Returns a string representation of the board rotations.

        Returns:
            str: Board formatted with only rotations shown.
        """
        characters = ''
        for rank in reversed(Rank):

            for file in File:
                mask = 1 << (rank * 10 + file)
                rotation = self.get_rotation_on(mask)
                has_piece = bb_helpers.is_occupied(self.combined_all_bitboard, mask)

                if has_piece:
                    characters += f'{rotation.upper()}  '
                else:
                    characters += '.  '

            characters += '\n\n'

        return characters

    def initialise_hash(self):
        """
        Initialises the Zobrist hash for the current board state.
        """
        for piece in Piece:
            for colour in Colour:
                piece_bitboard = self.get_piece_bitboard(piece, colour)

            for occupied_bitboard in bb_helpers.occupied_squares(piece_bitboard):
                self._hasher.apply_piece_hash(occupied_bitboard, piece, colour)

    for bitboard in bb_helpers.loop_all_squares():
        rotation = self.get_rotation_on(bitboard)
        self._hasher.apply_rotation_hash(bitboard, rotation)

    if self.active_colour == Colour.RED:
        self._hasher.apply_red_move_hash()

def flip_colour(self):
    """
    Flips the active colour and updates the Zobrist hash.
    """
    self.active_colour = self.active_colour.get_flipped_colour()

    if self.active_colour == Colour.RED:
        self._hasher.apply_red_move_hash()

def update_move(self, src, dest):
    """
    Updates the bitboards for a move.

    Args:
        src (int): The bitboard representation of the source square.
        dest (int): The bitboard representation of the destination square.
    """
    piece = self.get_piece_on(src, self.active_colour)

    self.clear_square(src, Colour.BLUE)
    self.clear_square(dest, Colour.BLUE)
    self.clear_square(src, Colour.RED)
    self.clear_square(dest, Colour.RED)

    self.set_square(dest, piece, self.active_colour)

def update_rotation(self, src, dest, new_rotation):
    """
    Updates the rotation bitboards for a move.

    Args:
        src (int): The bitboard representation of the source square.
        dest (int): The bitboard representation of the destination square.
        new_rotation (Rotation): The new rotation.
    """
    self.clear_rotation(src)
    self.set_rotation(dest, new_rotation)

def clear_rotation(self, bitboard):
    """
    Clears the rotation for a given square.

    Args:
        bitboard (int): The bitboard representation of the square.
    """
    old_rotation = self.get_rotation_on(bitboard)
    rotation_1, rotation_2 = self.rotation_bitboards
    self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.clear_square(rotation_1, bitboard)
```

```python
        self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.clear_square(rotation_2, bitboard)

        self._hasher.apply_rotation_hash(bitboard, old_rotation)

    def clear_square(self, bitboard, colour):
        """
        Clears a square piece and rotation for a given colour.

        Args:
            bitboard (int): The bitboard representation of the square.
            colour (Colour): The colour to clear.
        """
        piece = self.get_piece_on(bitboard, colour)

        if piece is None:
            return

        piece_bitboard = self.get_piece_bitboard(piece, colour)
        colour_bitboard = self.combined_colour_bitboards[colour]
        all_bitboard = self.combined_all_bitboard

        self.piece_bitboards[colour][piece] = bb_helpers.clear_square(piece_bitboard, bitboard)
        self.combined_colour_bitboards[colour] = bb_helpers.clear_square(colour_bitboard, bitboard)
        self.combined_all_bitboard = bb_helpers.clear_square(all_bitboard, bitboard)

        self._hasher.apply_piece_hash(bitboard, piece, colour)

    def set_rotation(self, bitboard, rotation):
        """
        Sets the rotation for a given square.

        Args:
            bitboard (int): The bitboard representation of the square.
            rotation (Rotation): The rotation to set.
        """
        rotation_1, rotation_2 = self.rotation_bitboards
        self._hasher.apply_rotation_hash(bitboard, rotation)

        match rotation:
            case Rotation.UP:
                return
            case Rotation.RIGHT:
                self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.set_square(rotation_1, bitboard)
                return
            case Rotation.DOWN:
                self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.set_square(rotation_2, bitboard)
                return
            case Rotation.LEFT:
                self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.set_square(rotation_1, bitboard)
                self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.set_square(rotation_2, bitboard)
                return
            case _:
                raise ValueError('Invalid rotation input (bitboard.py):', rotation)

    def set_square(self, bitboard, piece, colour):
        """
        Sets a piece on a given square.

        Args:
            bitboard (int): The bitboard representation of the square.
            piece (Piece): The piece to set.
            colour (Colour): The colour of the piece.
        """
        piece_bitboard = self.get_piece_bitboard(piece, colour)
        colour_bitboard = self.combined_colour_bitboards[colour]
        all_bitboard = self.combined_all_bitboard

        self.piece_bitboards[colour][piece] = bb_helpers.set_square(piece_bitboard, bitboard)
        self.combined_colour_bitboards[colour] = bb_helpers.set_square(colour_bitboard, bitboard)
        self.combined_all_bitboard = bb_helpers.set_square(all_bitboard, bitboard)

        self._hasher.apply_piece_hash(bitboard, piece, colour)

    def get_piece_bitboard(self, piece, colour):
        """
        Gets the bitboard for a piece type for a given colour.

        Args:
            piece (Piece): The piece bitboard to get.
            colour (Colour): The colour of the piece.

        Returns:
            int: The bitboard representation for all squares occupied by that piece type.
        """
        return self.piece_bitboards[colour][piece]

    def get_piece_on(self, target_bitboard, colour):
        """
        Gets the piece on a given square for a given colour.

        Args:
            target_bitboard (int): The bitboard representation of the square.
            colour (Colour): The colour of the piece.

        Returns:
            Piece: The piece on the square, or None if square is empty.
        """
        if not (bb_helpers.is_occupied(self.combined_colour_bitboards[colour], target_bitboard)):
            return None

        return next(
            (piece for piece in Piece if
                bb_helpers.is_occupied(self.get_piece_bitboard(piece, colour), target_bitboard)),
            None)

    def get_rotation_on(self, target_bitboard):
        """
        Gets the rotation on a given square.

        Args:
            target_bitboard (int): The bitboard representation of the square.

        Returns:
            Rotation: The rotation on the square.
        """
        rotationBits = [bb_helpers.is_occupied(self.rotation_bitboards[RotationIndex.SECONDBIT], target_bitboard), bb_helpers.is_occupied(self.rotation_bitboards[RotationIndex.FIRSTBIT], target_bitboard)]

        match rotationBits:
            case [False, False]:
                return Rotation.UP
            case [False, True]:
                return Rotation.RIGHT
            case [True, False]:
                return Rotation.DOWN
            case [True, True]:
                return Rotation.LEFT

    def get_colour_on(self, target_bitboard):
        """
        Gets the colour of the piece on a given square.

        Args:
            target_bitboard (int): The bitboard representation of the square.

        Returns:
            Colour: The colour of the piece on the square.
        """
        for piece in Piece:
            if self.get_piece_bitboard(piece, Colour.BLUE) & target_bitboard != EMPTY_BB:
                return Colour.BLUE
            elif self.get_piece_bitboard(piece, Colour.RED) & target_bitboard != EMPTY_BB:
                return Colour.RED

    def get_piece_count(self, piece, colour):
        """
        Gets the count of a given piece type and colour.

        Args:
            piece (Piece): The piece to count.
            colour (Colour): The colour of the piece.

        Returns:
            int: The number of that piece of that colour on the board.
        """
        return bb_helpers.pop_count(self.get_piece_bitboard(piece, colour))

    def get_hash(self):
        """
        Gets the Zobrist hash of the current board state.

        Returns:
            int: The Zobrist hash.
        """
        return self._hasher.hash

    def convert_to_piece_list(self):
        """
        Converts all bitboards to a list of pieces.
```

```python

        Returns:
            list: Board represented as a 2D list of Piece and Rotation objects.
        """
        piece_list = []

        for i in range(80):
            if x := self.get_piece_on(1 << i, Colour.BLUE):
                rotation = self.get_rotation_on(1 << i)
                piece_list.append((x.upper(), rotation))
            elif y := self.get_piece_on(1 << i, Colour.RED):
                rotation = self.get_rotation_on(1 << i)
                piece_list.append((y, rotation))
            else:
                piece_list.append(None)

        return piece_list
```

```python
from data.states.game.components.move import Move
from data.states.game.components.laser import Laser

from data.constants import Colour, Piece, Rank, File, MoveType, RotationDirection, Miscellaneous, A_FILE_MASK, J_FILE_MASK, ONE_RANK_MASK, EIGHT_RANK_MASK, EMPTY_BB
from data.states.game.components.bitboard_collection import BitboardCollection
from data.utils import bitboard_helpers as bb_helpers
from collections import defaultdict

class Board:
    def __init__(self, fen_string="sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b"):
        self.bitboards = BitboardCollection(fen_string)
        self.hash_list = [self.bitboards.get_hash()]

    def __str__(self):
        """
        Returns a string representation of the board.

        Returns:
            str: Board formatted as string.
        """
        characters = '8  '
        pieces = defaultdict(int)

        for rank_idx, rank in enumerate(reversed(Rank)):
            for file_idx, file in enumerate(File):
                mask = 1 << (rank * 10 + file)
                blue_piece = self.bitboards.get_piece_on(mask, Colour.BLUE)
                red_piece = self.bitboards.get_piece_on(mask, Colour.RED)

                if blue_piece:
                    pieces[blue_piece.value.upper()] += 1
                    characters += f'{blue_piece.upper()}  '
                elif red_piece:
                    pieces[red_piece.value] += 1
                    characters += f'{red_piece}  '
                else:
                    characters += '.  '

            characters += f'\n\n{7 - rank_idx}  '
        characters += 'A  B  C  D  E  F  G  H  I  J\n\n'
        characters += str(dict(pieces))
        characters += f'\nCURRENT PLAYER TO MOVE: {self.bitboards.active_colour.name}\n'
        return characters

    def get_piece_list(self):
        """
        Converts the board bitboards to a list of pieces.

        Returns:
            list: List of Pieces.
        """
        return self.bitboards.convert_to_piece_list()

    def get_active_colour(self):
        """
        Gets the active colour.

        Returns:
            Colour: The active colour.
        """
        return self.bitboards.active_colour

    def to_hash(self):
        """
        Gets the hash of the current board state.

        Returns:
            int: A Zobrist hash.
        """
        return self.bitboards.get_hash()

    def check_win(self):
        """
        Checks for a Pharoah capture or threefold-repetition.

        Returns:
            Colour | Miscellaneous: The winning colour, or Miscellaneous.DRAW.
        """
        for colour in Colour:
            if self.bitboards.get_piece_bitboard(Piece.PHAROAH, colour) == EMPTY_BB:
                return colour.get_flipped_colour()

        if self.hash_list.count(self.hash_list[-1]) >= 3:
            return Miscellaneous.DRAW

        return None

    def apply_move(self, move, fire_laser=True, add_hash=False):
        """
        Applies a move to the board.

        Args:
            move (Move): The move to apply.
            fire_laser (bool): Whether to fire the laser after the move.
            add_hash (bool): Whether to add the board state hash to the hash list.

        Returns:
            Laser: The laser trajectory result.
        """
        piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.active_colour)

        if piece_symbol is None:
            raise ValueError('Invalid move - no piece found on source square')
        elif piece_symbol == Piece.SPHINX:
            raise ValueError('Invalid move - sphinx piece is immovable')

        if move.move_type == MoveType.MOVE:
            possible_moves = self.get_valid_squares(move.src)
            if bb_helpers.is_occupied(move.dest, possible_moves) is False:
                raise ValueError('Invalid move - destination square is occupied')

            piece_rotation = self.bitboards.get_rotation_on(move.src)

            self.bitboards.update_move(move.src, move.dest)
            self.bitboards.update_rotation(move.src, move.dest, piece_rotation)

        elif move.move_type == MoveType.ROTATE:
            piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.active_colour)
            piece_rotation = self.bitboards.get_rotation_on(move.src)

            if move.rotation_direction == RotationDirection.CLOCKWISE:
                new_rotation = piece_rotation.get_clockwise()
            elif move.rotation_direction == RotationDirection.ANTICLOCKWISE:
                new_rotation = piece_rotation.get_anticlockwise()

            self.bitboards.update_rotation(move.src, move.src, new_rotation)

        laser = None
        if fire_laser:
            laser = self.fire_laser(add_hash)

        if add_hash:
            self.hash_list.append(self.bitboards.get_hash())

        self.bitboards.flip_colour()

        return laser

    def undo_move(self, move, laser_result):
        """
        Undoes a move on the board.

        Args:
            move (Move): The move to undo.
            laser_result (Laser): The laser trajectory result.
        """
        self.bitboards.flip_colour()

        if laser_result.hit_square_bitboard:
            # Get info of destroyed piece, and add it to the board again
            src = laser_result.hit_square_bitboard
            piece = laser_result.piece_hit
            colour = laser_result.piece_colour
            rotation = laser_result.piece_rotation

            self.bitboards.set_square(src, piece, colour)
            self.bitboards.clear_rotation(src)
            self.bitboards.set_rotation(src, rotation)

        # Create new Move object that is the inverse of the passed move
        if move.move_type == MoveType.MOVE:
            reversed_move = Move.instance_from_bitboards(MoveType.MOVE, move.dest, move.src)
        elif move.move_type == MoveType.ROTATE:
            reversed_move = Move.instance_from_bitboards(MoveType.ROTATE, move.src, move.src, move.rotation_direction.get_opposite())

        self.apply_move(reversed_move, fire_laser=False)
        self.bitboards.flip_colour()

    def remove_piece(self, square_bitboard):
        """
        Removes a piece from a given square.

        Args:
            square_bitboard (int): The bitboard representation of the square.
        """
        self.bitboards.clear_square(square_bitboard, Colour.BLUE)
        self.bitboards.clear_square(square_bitboard, Colour.RED)
        self.bitboards.clear_rotation(square_bitboard)

    def get_valid_squares(self, src_bitboard, colour=None):
        """
        Gets valid squares for a piece to move to.

        Args:
            src_bitboard (int): The bitboard representation of the source square.
            colour (Colour, optional): The active colour of the piece.

        Returns:
            int: The bitboard representation of valid squares.
        """
        target_top_left = (src_bitboard & A_FILE_MASK & EIGHT_RANK_MASK) << 9
        target_top_middle = (src_bitboard & EIGHT_RANK_MASK) << 10
        target_top_right = (src_bitboard & J_FILE_MASK & EIGHT_RANK_MASK) << 11
        target_middle_right = (src_bitboard & J_FILE_MASK) << 1

        target_bottom_right = (src_bitboard & J_FILE_MASK & ONE_RANK_MASK) >> 9
        target_bottom_middle = (src_bitboard & ONE_RANK_MASK) >> 10
        target_bottom_left = (src_bitboard & A_FILE_MASK & ONE_RANK_MASK)>> 11
        target_middle_left = (src_bitboard & A_FILE_MASK) >> 1

        possible_moves = target_top_left | target_top_middle | target_top_right | target_middle_right | target_bottom_right | target_bottom_middle | target_bottom_left | target_middle_left

        if colour is not None:
            valid_possible_moves = possible_moves & ~self.bitboards.combined_colour_bitboards[colour]
        else:
            valid_possible_moves = possible_moves & ~self.bitboards.combined_all_bitboard

        return valid_possible_moves

    def get_mobility(self, colour):
        """
        Gets all valid squares for a given colour.

        Args:
            colour (Colour): The colour of the pieces.

        Returns:
            int: The bitboard representation of all valid squares.
        """
        active_pieces = self.get_all_active_pieces(colour)
        possible_moves = 0

        for square in bb_helpers.occupied_squares(active_pieces):
            possible_moves += bb_helpers.pop_count(self.get_valid_squares(square))

        return possible_moves

    def get_all_active_pieces(self, colour=None):
        """
        Gets all active pieces for the current player.

        Args:
            colour (Colour): Active colour of pieces to retrieve. Defaults to None.

        Returns:
            int: The bitboard representation of all active pieces.
        """
        if colour is None:
            colour = self.bitboards.active_colour

        active_pieces = self.bitboards.combined_colour_bitboards[colour]
        sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
        return active_pieces ^ sphinx_bitboard

    def fire_laser(self, remove_hash):
        """
        Fires the laser and removes hit pieces.

        Args:
            remove_hash (bool): Whether to clear the hash list if a piece is hit.

        Returns:
            Laser: The result of firing the laser.
        """
        laser = Laser(self.bitboards)

        if laser.hit_square_bitboard:
            self.remove_piece(laser.hit_square_bitboard)

            if remove_hash:
                self.hash_list = [] # Remove all hashes for threefold repetition, as the position is impossible to be repeated after a piece is removed
        return laser

    def generate_square_moves(self, src):
        """
        Generates all valid moves for a piece on a given square.

        Args:
            src (int): The bitboard representation of the source square.

        Yields:
            Move: A valid move for the piece.
        """
        for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
            yield Move(MoveType.MOVE, src, dest)

    def generate_all_moves(self, colour):
```

```python
        """
        Generates all valid moves for a given colour.

        Args:
            colour (Colour): The colour of the pieces.

        Yields:
            Move: A valid move for the active colour.
        """
        sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
        # Remove source squares for Sphinx pieces, as they cannot be moved
        sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour] ^ sphinx_bitboard

        for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
            # Generate movement moves
            yield from self.generate_square_moves(square)

            # Generate rotational moves
            for rotation_direction in RotationDirection:
                yield Move(MoveType.ROTATE, square, rotation_direction=rotation_direction)
```

```python
from data.states.game.components.particles_draw import ParticlesDraw
from data.utils.board_helpers import coords_to_screen_pos
from data.constants import Colour, ShaderType
from data.managers.window import window
from data.managers.animation import animation

class CaptureDraw:
    def __init__(self, board_position, board_size):
        self._board_position = board_position
        self._square_size = board_size[0] / 10
        self._particles_draw = ParticlesDraw()

    def add_capture(self, piece, colour, rotation, piece_coords, sphinx_coords, active_colour, particles=True, shake=True):
        if particles:
            self._particles_draw.add_captured_piece(
                piece,
                colour,
                rotation,
                coords_to_screen_pos(piece_coords, self._board_position, self._square_size),
                self._square_size
            )
            self._particles_draw.add_sparks(
                3,
                (255, 0, 0) if active_colour == Colour.RED else (0, 0, 255),
                coords_to_screen_pos(sphinx_coords, self._board_position, self._square_size)
            )

        if shake:
            window.set_effect(ShaderType.SHAKE)
            animation.set_timer(500, lambda: window.clear_effect(ShaderType.SHAKE))

    def draw(self, screen):
        self._particles_draw.draw(screen)

    def update(self):
        self._particles_draw.update()

    def handle_resize(self, board_position, board_size):
        self._board_position = board_position
        self._square_size = board_size[0] / 10
```

```python
import pygame
from data.constants import CursorMode
from data.states.game.components.piece_sprite import PieceSprite
from data.managers.cursor import cursor
from data.managers.audio import audio
from data.assets import SFX

DRAG_THRESHOLD = 500

class DragAndDrop:
    def __init__(self, board_position, board_size, change_cursor=True):
        self._board_position = board_position
        self._board_size = board_size
        self._change_cursor = change_cursor
        self._ticks_since_drag = 0

        self.dragged_sprite = None

    def set_dragged_piece(self, piece, colour, rotation):
        sprite = PieceSprite(piece=piece, colour=colour, rotation=rotation)
        sprite.set_geometry((0, 0), self._board_size[0] / 10)
        sprite.set_image()

        self.dragged_sprite = sprite
        self._ticks_since_drag = pygame.time.get_ticks()

        if self._change_cursor:
            cursor.set_mode(CursorMode.CLOSEDHAND)

    def remove_dragged_piece(self):
        self.dragged_sprite = None
        time_dragged = pygame.time.get_ticks() - self._ticks_since_drag
        self._ticks_since_drag = 0

        if self._change_cursor:
            cursor.set_mode(CursorMode.OPENHAND)

        return time_dragged > DRAG_THRESHOLD

    def get_dragged_info(self):
        return self.dragged_sprite.type, self.dragged_sprite.colour, self.
    dragged_sprite.rotation

    def draw(self, screen):
        if self.dragged_sprite is None:
            return

        self.dragged_sprite.rect.center = pygame.mouse.get_pos()
        screen.blit(self.dragged_sprite.image, self.dragged_sprite.rect.topleft)

    def handle_resize(self, board_position, board_size):
        if self.dragged_sprite:
            self.dragged_sprite.set_geometry(board_position, board_size[0] / 10)

        self._board_position = board_position
        self._board_size = board_size
```

```python
from data.constants import Colour, RotationIndex, Rotation, Piece, EMPTY_BB
from data.utils.bitboard_helpers import occupied_squares, print_bitboard, bitboard_to_index

def parse_fen_string(fen_string):
    #sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b
    piece_bitboards = [{char: EMPTY_BB for char in Piece}, {char: EMPTY_BB for char in Piece}]
    rotation_bitboards = [EMPTY_BB, EMPTY_BB]
    combined_colour_bitboards = [EMPTY_BB, EMPTY_BB]
    combined_all_bitboard = 0
    part_1, part_2 = fen_string.split(' ')

    rank = 7
    file = 0

    piece_count = {char.lower(): 0 for char in Piece} | {char.upper(): 0 for char in Piece}

    for index, character in enumerate(part_1):
        square = rank * 10 + file

        if character.lower() in Piece:
            piece_count[character] += 1
            if character.isupper():
                piece_bitboards[Colour.BLUE][character.lower()] |= 1 << square

            else:
                piece_bitboards[Colour.RED][character.lower()] |= 1 << square

            rotation = part_1[index + 1]
            match rotation:
                case Rotation.UP:
                    pass
                case Rotation.RIGHT:
                    rotation_bitboards[RotationIndex.FIRSTBIT] |= 1 << square
                case Rotation.DOWN:
                    rotation_bitboards[RotationIndex.SECONDBIT] |= 1 << square
                case Rotation.LEFT:
                    rotation_bitboards[RotationIndex.SECONDBIT] |= 1 << square
                    rotation_bitboards[RotationIndex.FIRSTBIT] |= 1 << square
                case _:
                    raise ValueError('Invalid FEN String - piece character not followed by rotational character')

            file += 1
        elif character in '0123456789':
            if character == '1' and fen_string[index + 1] == '0':
                file += 10
                continue

            file += int(character)
        elif character == '/':
            rank = rank - 1
            file = 0
        elif character in Rotation:
            continue
        else:
            raise ValueError('Invalid FEN String - invalid character found:', character)

    if piece_count['s'] != 1 or piece_count['S'] != 1:
        raise ValueError('Invalid FEN string - invalid number of Sphinx pieces')
    # COMMENTED OUT AS NO PHAROAH PIECES IS OKAY IF PARSING FEN STRING FOR FINISHED GAME BOARD THUMBNAIL
    elif piece_count['f'] > 1 or piece_count['F'] > 1:
        raise ValueError('Invalid FEN string - invalid number of Pharoah pieces')

    if part_2 == 'b':
        colour = Colour.BLUE
    elif part_2 == 'r':
        colour = Colour.RED
    else:
        raise ValueError('Invalid FEN string - invalid active colour')

    for piece in Piece:
        combined_colour_bitboards[Colour.BLUE] |= piece_bitboards[Colour.BLUE][piece]
        combined_colour_bitboards[Colour.RED] |= piece_bitboards[Colour.RED][piece]

    combined_all_bitboard = combined_colour_bitboards[Colour.BLUE] | combined_colour_bitboards[Colour.RED]
    return (piece_bitboards, combined_colour_bitboards, combined_all_bitboard, rotation_bitboards, colour)

def encode_fen_string(bitboard_collection):
    blue_bitboards = bitboard_collection.piece_bitboards[Colour.BLUE]
    red_bitboards = bitboard_collection.piece_bitboards[Colour.RED]

    fen_string_list = [''] * 80

    for piece, bitboard in blue_bitboards.items():
        for individual_bitboard in occupied_squares(bitboard):
            index = bitboard_to_index(individual_bitboard)
            rotation = bitboard_collection.get_rotation_on(individual_bitboard)
            fen_string_list[index] = piece.upper() + rotation

    for piece, bitboard in red_bitboards.items():
        for individual_bitboard in occupied_squares(bitboard):
            index = bitboard_to_index(individual_bitboard)
            rotation = bitboard_collection.get_rotation_on(individual_bitboard)
            fen_string_list[index] = piece.lower() + rotation

    fen_string = ''
    row_string = ''
    empty_count = 0
    for index, square in enumerate(fen_string_list):
        if square == '':
            empty_count += 1
        else:
            if empty_count > 0:
                row_string += str(empty_count)
                empty_count = 0

            row_string += square

        if index % 10 == 9:
            if empty_count > 0:
                fen_string = '/' + row_string + str(empty_count) + fen_string
            else:
                fen_string = '/' + row_string + fen_string

            row_string = ''
            empty_count = 0

    fen_string = fen_string[1:]

    if bitboard_collection.active_colour == Colour.BLUE:
        colour = 'b'
    else:
        colour = 'r'

    return fen_string + ' ' + colour
```

```
from data.utils import bitboard_helpers as bb_helpers
from data.constants import Piece, Colour, Rotation, A_FILE_MASK, J_FILE_MASK,
    ONE_RANK_MASK, EIGHT_RANK_MASK, EMPTY_BB
from data.utils.bitboard_helpers import print_bitboard

class Laser:
    def __init__(self, bitboards):
        self._bitboards = bitboards
        self.hit_square_bitboard, self.piece_hit, self.laser_path, self.
    path_bitboard, self.pieces_on_trajectory = self.calculate_trajectory()

        if (self.hit_square_bitboard != EMPTY_BB):
            self.piece_rotation = self._bitboards.get_rotation_on(self.
    hit_square_bitboard)
            self.piece_colour = self._bitboards.get_colour_on(self.
    hit_square_bitboard)

    def calculate_trajectory(self):
        current_square = self._bitboards.get_piece_bitboard(Piece.SPHINX, self.
    _bitboards.active_colour)
        previous_direction = self._bitboards.get_rotation_on(current_square)
        trajectory_bitboard = 0b0
        trajectory_list = []
        square_animation_states = []
        pieces_on_trajectory = []

        while current_square:
            current_piece = self._bitboards.get_piece_on(current_square, Colour.
    BLUE) or self._bitboards.get_piece_on(current_square, Colour.RED)
            current_rotation = self._bitboards.get_rotation_on(current_square)

            next_square, direction, piece_hit = self.calculate_next_square(
    current_square, current_piece, current_rotation, previous_direction)

            trajectory_bitboard |= current_square
            trajectory_list.append(bb_helpers.bitboard_to_coords(current_square))
            square_animation_states.append(direction)

            if previous_direction != direction:
                pieces_on_trajectory.append(current_square)

            if next_square == EMPTY_BB:
                hit_square_bitboard = 0b0

                if piece_hit:
                    hit_square_bitboard = current_square

                return hit_square_bitboard, piece_hit, list(zip(trajectory_list,
    square_animation_states)), trajectory_bitboard, pieces_on_trajectory

```

```
            current_square = next_square
            previous_direction = direction

    def calculate_next_square(self, square, piece, rotation, previous_direction):
        match piece:
            case Piece.SPHINX:
                if previous_direction != rotation:
                    return EMPTY_BB, previous_direction, None

                next_square = self.next_square_bitboard(square, rotation)
                return next_square, previous_direction, Piece.SPHINX

            case Piece.PYRAMID:
                if previous_direction in [rotation, rotation.get_clockwise()]:
                    return EMPTY_BB, previous_direction, Piece.PYRAMID

                if previous_direction == rotation.get_anticlockwise():
                    new_direction = previous_direction.get_clockwise()
                else:
                    new_direction = previous_direction.get_anticlockwise()

                next_square = self.next_square_bitboard(square, new_direction)

                return next_square, new_direction, None

            case Piece.ANUBIS:
                if previous_direction == rotation.get_clockwise().get_clockwise():
                    return EMPTY_BB, previous_direction, None

                return EMPTY_BB, previous_direction, Piece.ANUBIS

            case Piece.SCARAB:
                if previous_direction in [rotation.get_clockwise(), rotation.get_anticlockwise()]:
                    new_direction = previous_direction.get_anticlockwise()
                else:
                    new_direction = previous_direction.get_clockwise()

                next_square = self.next_square_bitboard(square, new_direction)

                return next_square, new_direction, None

            case Piece.PHAROAH:
                return EMPTY_BB, previous_direction, Piece.PHAROAH

            case None:
                next_square = self.next_square_bitboard(square, previous_direction)

                return next_square, previous_direction, None

    def next_square_bitboard(self, src_bitboard, previous_direction):
        match previous_direction:
            case Rotation.UP:
                masked_src_bitboard = src_bitboard & EIGHT_RANK_MASK
                return masked_src_bitboard << 10
            case Rotation.RIGHT:
                masked_src_bitboard = src_bitboard & J_FILE_MASK
                return masked_src_bitboard << 1
            case Rotation.DOWN:
                masked_src_bitboard = src_bitboard & ONE_RANK_MASK
                return masked_src_bitboard >> 10
            case Rotation.LEFT:
                masked_src_bitboard = src_bitboard & A_FILE_MASK
                return masked_src_bitboard >> 1
```

```
import pygame
from data.utils.board_helpers import coords_to_screen_pos
from data.constants import EMPTY_BB, ShaderType, Colour
from data.managers.animation import animation
from data.managers.window import window
from data.managers.audio import audio
from data.assets import GRAPHICS, SFX
from data.constants import LaserType

type_to_image = {
    LaserType.END: ['laser_end_1', 'laser_end_2'],
    LaserType.STRAIGHT: ['laser_straight_1', 'laser_straight_2'],
    LaserType.CORNER: ['laser_corner_1', 'laser_corner_2']
}

GLOW_SCALE_FACTOR = 1.5

class LaserDraw:
    def __init__(self, board_position, board_size):
        self._board_position = board_position
        self._square_size = board_size[0] / 10
        self._laser_lists = []

    @property
    def firing(self):
        return len(self._laser_lists) > 0

    def add_laser(self, laser_result, laser_colour):
        """
        Adds a laser to the board.

        Args:
            laser_result (Laser): Laser class instance containing laser trajectory
     info.
            laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
        """
        laser_path = laser_result.laser_path.copy()
        laser_types = [LaserType.END]
        # List of angles in degree to rotate the laser image surface when drawn
        laser_rotation = [laser_path[0][1]]
        laser_lights = []

        # Iterates through every square laser passes through
        for i in range(1, len(laser_path)):
            previous_direction = laser_path[i-1][1]
            current_coords, current_direction = laser_path[i]

            if current_direction == previous_direction:
                laser_types.append(LaserType.STRAIGHT)
                laser_rotation.append(current_direction)
            elif current_direction == previous_direction.get_clockwise():
                laser_types.append(LaserType.CORNER)
                laser_rotation.append(current_direction)
            elif current_direction == previous_direction.get_anticlockwise():
                laser_types.append(LaserType.CORNER)
                laser_rotation.append(current_direction.get_anticlockwise())

            # Adds a shader ray effect on the first and last square of the laser
```

```
trajectory
        if i in [1, len(laser_path) - 1]:
            abs_position = coords_to_screen_pos(current_coords, self._board_position, self._square_size)
            laser_lights.append([
                (abs_position[0] / window.size[0], abs_position[1] / window.size[1]),
                0.35,
                (0, 0, 255) if laser_colour == Colour.BLUE else (255, 0, 0),
            ])

    # Sets end laser draw type if laser hits a piece
    if laser_result.hit_square_bitboard != EMPTY_BB:
        laser_types[-1] = LaserType.END
        laser_path[-1] = (laser_path[-1][0], laser_path[-2][1].get_opposite())
        laser_rotation[-1] = laser_path[-2][1].get_opposite()

        audio.play_sfx(SFX['piece_destroy'])

    laser_path = [(coords, rotation, type) for (coords, dir), rotation, type in zip(laser_path, laser_rotation, laser_types)]
    self._laser_lists.append((laser_path, laser_colour))

    window.clear_effect(ShaderType.RAYS)
    window.set_effect(ShaderType.RAYS, lights=laser_lights)
    animation.set_timer(1000, self.remove_laser)

    audio.play_sfx(SFX['laser_1'])
    audio.play_sfx(SFX['laser_2'])

def remove_laser(self):
    """
    Removes a laser from the board.
    """
    self._laser_lists.pop(0)

    if len(self._laser_lists) == 0:
        window.clear_effect(ShaderType.RAYS)

def draw_laser(self, screen, laser_list, glow=True):
    """
    Draws every laser on the screen.

    Args:
        screen (pygame.Surface): The screen to draw on.
        laser_list (list): The list of laser segments to draw.
        glow (bool, optional): Whether to draw a glow effect. Defaults to True.
    """
    laser_path, laser_colour = laser_list
    laser_list = []
    glow_list = []

    for coords, rotation, type in laser_path:
        square_x, square_y = coords_to_screen_pos(coords, self._board_position, self._square_size)

        image = GRAPHICS[type_to_image[type][laser_colour]]
        rotated_image = pygame.transform.rotate(image, rotation.to_angle())
        scaled_image = pygame.transform.scale(rotated_image, (self._square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating black lines
```

```python
            laser_list.append((scaled_image, (square_x, square_y)))

            # Scales up the laser image surface as a glow surface
            scaled_glow = pygame.transform.scale(rotated_image, (self._square_size * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
            offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
            glow_list.append((scaled_glow, (square_x - offset, square_y - offset)))

        # Scaled glow surfaces drawn on top with the RGB_ADD blend mode
        if glow:
            screen.fblits(glow_list, pygame.BLEND_RGB_ADD)

        screen.blits(laser_list)

    def draw(self, screen):
        """
        Draws all lasers on the screen.

        Args:
            screen (pygame.Surface): The screen to draw on.
        """
        for laser_list in self._laser_lists:
            self.draw_laser(screen, laser_list)

    def handle_resize(self, board_position, board_size):
        """
        Handles resizing of the board.

        Args:
            board_position (tuple[int, int]): The new position of the board.
            board_size (tuple[int, int]): The new size of the board.
        """
        self._board_position = board_position
        self._square_size = board_size[0] / 10
```

```python
from data.constants import MoveType, Colour, RotationDirection
from data.utils.bitboard_helpers import notation_to_bitboard, coords_to_bitboard, bitboard_to_coords, bitboard_to_notation, print_bitboard
import re
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class Move():
    def __init__(self, move_type, src, dest=None, rotation_direction=None):
        self.move_type = move_type
        self.src = src
        self.dest = dest
        self.rotation_direction = rotation_direction

    def to_notation(self, colour, piece, hit_square_bitboard):
        hit_square = ''
        if colour == Colour.BLUE:
            piece = piece.upper()

        if hit_square_bitboard:
            hit_square = 'x' + bitboard_to_notation(hit_square_bitboard)

        if self.move_type == MoveType.MOVE:
            return 'M' + piece + bitboard_to_notation(self.src) + bitboard_to_notation(self.dest) + hit_square
        else:
            return 'R' + piece + bitboard_to_notation(self.src) + self.rotation_direction + hit_square

    def __str__(self):
        rotate_text = ''
        coords_1 = '(' + chr(bitboard_to_coords(self.src)[0] + 65) + ',' + str(bitboard_to_coords(self.src)[1] + 1) + ')'

        if self.move_type == MoveType.ROTATE:
            rotate_text = ' ' + self.rotation_direction.name
            return f'{self.move_type.name}{rotate_text}: ON {coords_1}'

        elif self.move_type == MoveType.MOVE:
            coords_2 = '(' + chr(bitboard_to_coords(self.dest)[0] + 65) + ', ' + str(bitboard_to_coords(self.dest)[1] + 1) + ')'
            return f'{self.move_type.name}{rotate_text}: FROM {coords_1} TO {coords_2}'

        # (Rotation: {self.rotation_direction})

    @classmethod
    def instance_from_notation(move_cls, notation):
        try:
            notation = notation.split('x')[0]
            move_type = notation[0].lower()

            moves = notation[2:]
            letters = re.findall(r'[A-Za-z]+', moves)
            numbers = re.findall(r'\d+', moves)

            if move_type == MoveType.MOVE:
                src_bitboard = notation_to_bitboard(letters[0] + numbers[0])
                dest_bitboard = notation_to_bitboard(letters[1] + numbers[1])

                return move_cls(move_type, src_bitboard, dest_bitboard)

            elif move_type == MoveType.ROTATE:
                src_bitboard = notation_to_bitboard(letters[0] + numbers[0])
                rotation_direction = RotationDirection(letters[1])

                return move_cls(move_type, src_bitboard, src_bitboard, rotation_direction)
            else:
                raise ValueError('(Move.instance_from_notation) Invalid move type: ', move_type)

        except Exception as error:
            logger.info('(Move.instance_from_notation) Error occured while parsing:', error)
            raise error

    @classmethod
    def instance_from_input(move_cls, move_type, src, dest=None, rotation=None):
        try:
            if move_type == MoveType.MOVE:
                src_bitboard = notation_to_bitboard(src)
                dest_bitboard = notation_to_bitboard(dest)

            elif move_type == MoveType.ROTATE:
                src_bitboard = notation_to_bitboard(src)
                dest_bitboard = src_bitboard

            return move_cls(move_type, src_bitboard, dest_bitboard, rotation)
        except Exception as error:
            logger.info('Error (Move.instance_from):', error)
            raise error

    @classmethod
    def instance_from_coords(move_cls, move_type, src_coords, dest_coords=None, rotation_direction=None):
        try:
            src_bitboard = coords_to_bitboard(src_coords)
            dest_bitboard = coords_to_bitboard(dest_coords)

            return move_cls(move_type, src_bitboard, dest_bitboard, rotation_direction)
        except Exception as error:
            logger.info('Error (Move.instance_from_coords):', error)
            raise error

    @classmethod
    def instance_from_bitboards(move_cls, move_type, src_bitboard, dest_bitboard=None, rotation_direction=None):
        try:
            return move_cls(move_type, src_bitboard, dest_bitboard, rotation_direction)
        except Exception as error:
            logger.info('Error (Move.instance_from_bitboards):', error)
            raise error
```

```python
import pygame
from data.constants import OVERLAY_COLOUR_LIGHT, OVERLAY_COLOUR_DARK
from data.utils.board_helpers import coords_to_screen_pos, screen_pos_to_coords, create_square_overlay, create_circle_overlay

class OverlayDraw:
    def __init__(self, board_position, board_size, limit_hover=True):
        self._board_position = board_position
        self._board_size = board_size

        self._hovered_coords = None
        self._selected_coords = None
        self._available_coords = None

        self._limit_hover = limit_hover

        self._selected_overlay = None
        self._hovered_overlay = None
        self._available_overlay = None

        self.initialise_overlay_surfaces()

    @property
    def square_size(self):
        return self._board_size[0] / 10

    def initialise_overlay_surfaces(self):
        self._selected_overlay = create_square_overlay(self.square_size, OVERLAY_COLOUR_DARK)
        self._hovered_overlay = create_square_overlay(self.square_size, OVERLAY_COLOUR_LIGHT)
        self._available_overlay = create_circle_overlay(self.square_size, OVERLAY_COLOUR_LIGHT)

    def set_hovered_coords(self, mouse_pos):
        self._hovered_coords = screen_pos_to_coords(mouse_pos, self._board_position, self._board_size)

    def set_selected_coords(self, coords):
        self._selected_coords = coords

    def set_available_coords(self, coords_list):
        self._available_coords = coords_list

    def set_hover_limit(self, new_limit):
        self._limit_hover = new_limit

    def draw(self, screen):
        self.set_hovered_coords(pygame.mouse.get_pos())

        if self._selected_coords:
            screen.blit(self._selected_overlay, coords_to_screen_pos(self._selected_coords, self._board_position, self.square_size))

        if self._available_coords:
            for coords in self._available_coords:
                screen.blit(self._available_overlay, coords_to_screen_pos(coords, self._board_position, self.square_size))

        if self._hovered_coords:
            if self._hovered_coords is None:
                return

            if self._limit_hover and ((self._available_coords is None) or (self._hovered_coords not in self._available_coords)):
                return

            screen.blit(self._hovered_overlay, coords_to_screen_pos(self._hovered_coords, self._board_position, self.square_size))

    def handle_resize(self, board_position, board_size):
        self._board_position = board_position
        self._board_size = board_size

        self.initialise_overlay_surfaces()
```

```python
import pygame
from random import randint
from data.utils.asset_helpers import get_perimeter_sample, get_vector, get_angle_between_vectors, get_next_corner
from data.states.game.components.piece_sprite import PieceSprite

class ParticlesDraw:
    def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
        self._particles = []
        self._glow_particles = []

        self._gravity = gravity
        self._rotation = rotation
        self._shrink = shrink
        self._opacity = opacity

    def fragment_image(self, image, number):
        image_size = image.get_rect().size
        """
    1. Takes an image surface and samples random points on the perimeter.
    2. Iterates through points, and depending on the nature of two consecutive points, finds a corner between them.
    3. Draws a polygon with the points as the vertices to mask out the area not in the fragment.

    Args:
        image (pygame.Surface): Image to fragment.
        number (int): The number of fragments to create.

    Returns:
        list[pygame.Surface]: List of image surfaces with fragment of original surface drawn on top.
    """
    center = image.get_rect().center
    points_list = get_perimeter_sample(image_size, number)
    fragment_list = []

    points_list.append(points_list[0])

    # Iterate through points_list, using the current point and the next one
    for i in range(len(points_list) - 1):
        vertex_1 = points_list[i]
        vertex_2 = points_list[i + 1]
        vector_1 = get_vector(center, vertex_1)
        vector_2 = get_vector(center, vertex_2)
        angle = get_angle_between_vectors(vector_1, vector_2)

        cropped_image = pygame.Surface(image_size, pygame.SRCALPHA)
        cropped_image.fill((0, 0, 0, 0))
        cropped_image.blit(image, (0, 0))

        corners_to_draw = None

        if vertex_1[0] == vertex_2[0] or vertex_1[1] == vertex_2[1]: # Points on the same side
            corners_to_draw = 4

        elif abs(vertex_1[0] - vertex_2[0]) == image_size[0] or abs(vertex_1[1] - vertex_2[1]) == image_size[1]: # Points on opposite sides
            corners_to_draw = 2

        elif angle < 180: # Points on adjacent sides
            corners_to_draw = 3

        else:
            corners_to_draw = 1

        corners_list = []
        for j in range(corners_to_draw):
            if len(corners_list) == 0:
                corners_list.append(get_next_corner(vertex_2, image_size))
            else:
                corners_list.append(get_next_corner(corners_list[-1], image_size))

        pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *corners_list, vertex_1))

        fragment_list.append(cropped_image)

    return fragment_list

    def add_captured_piece(self, piece, colour, rotation, position, size):
        """
        Adds a captured piece to fragment into particles.

        Args:
            piece (Piece): The piece type.
            colour (Colour): The active colour of the piece.
            rotation (int): The rotation of the piece.
            position (tuple[int, int]): The position where particles originate from.
            size (tuple[int, int]): The size of the piece.
        """
        piece_sprite = PieceSprite(piece, colour, rotation)
        piece_sprite.set_geometry((0, 0), size)
        piece_sprite.set_image()

        particles = self.fragment_image(piece_sprite.image, 5)

        for particle in particles:
            self.add_particle(particle, position)

    def add_sparks(self, radius, colour, position):
        """
        Adds laser spark particles.

        Args:
            radius (int): The radius of the sparks.
            colour (Colour): The active colour of the sparks.
            position (tuple[int, int]): The position where particles originate from.
        """
        for i in range(randint(10, 15)):
            velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
            random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in colour]
            self._particles.append([None, [radius, random_colour], [*position], velocity, 0])

    def add_particle(self, image, position):
        """
        Adds a particle.

        Args:
            image (pygame.Surface): The image of the particle.
            position (tuple): The position of the particle.
        """
        velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]

        # Each particle is stored with its attributes: [surface, copy of surface, position, velocity, lifespan]
        self._particles.append([image, image.copy(), [*position], velocity, 0])

    def update(self):
        """
        Updates each particle and its attributes.
        """
        for i in range(len(self._particles) - 1, -1, -1):
            particle = self._particles[i]

            #update position
            particle[2][0] += particle[3][0]
```

```python
            particle[2][1] += particle[3][1]

            #update lifespan
            self._particles[i][4] += 0.01

            if self._particles[i][4] >= 1:
                self._particles.pop(i)
                continue

            if isinstance(particle[1], pygame.Surface): # Particle is a piece
                # Update velocity
                particle[3][1] += self._gravity

                # Update size
                image_size = particle[1].get_rect().size
                end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink) * image_size[1])
                target_size = (image_size[0] - particle[4] * (image_size[0] - end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))

                # Update rotation
                rotation = (self._rotation if particle[3][0] <= 0 else -self._rotation) * particle[4]

                updated_image = pygame.transform.scale(pygame.transform.rotate(particle[1], rotation), target_size)

            elif isinstance(particle[1], list): # Particle is a spark
                # Update size
                end_radius = (1 - self._shrink) * particle[1][0]
                target_radius = particle[1][0] - particle[4] * (particle[1][0] - end_radius)

                updated_image = pygame.Surface((target_radius * 2, target_radius * 2), pygame.SRCALPHA)
                pygame.draw.circle(updated_image, particle[1][1], (target_radius, target_radius), target_radius)

            # Update opacity
            alpha = 255 - particle[4] * (255 - self._opacity)

            updated_image.fill((255, 255, 255, alpha), None, pygame.BLEND_RGBA_MULT)

            particle[0] = updated_image

    def draw(self, screen):
        """
        Draws the particles, indexing the surface and position attributes for each particle.

        Args:
            screen (pygame.Surface): The screen to draw on.
        """
        screen.blits([
            (particle[0], particle[2]) for particle in self._particles
        ])
```

```python
import pygame
from data.constants import EMPTY_BB, Colour, Piece
from data.states.game.components.piece_sprite import PieceSprite
from data.utils.board_helpers import coords_to_screen_pos
from data.utils import bitboard_helpers as bb_helpers

class PieceGroup(pygame.sprite.Group):
    def __init__(self):
        # self.square_list = []
        # self.valid_square_list_positions = []
        super().__init__()

    def initialise_pieces(self, piece_list, board_position, board_size):
        self.empty()

        for index, piece_and_rotation in enumerate(piece_list):
            x = index % 10
            y = index // 10

            if piece_and_rotation:
                if piece_and_rotation[0].isupper():
                    colour = Colour.BLUE
                else:
                    colour = Colour.RED

                piece = PieceSprite(piece=Piece(piece_and_rotation[0].lower()), colour=colour, rotation=piece_and_rotation[1])
                piece.set_coords((x, y))
                piece.set_geometry(board_position, board_size[0] / 10)
                piece.set_image()
                self.add(piece)

    def set_geometry(self, board_position, board_size):
        for sprite in self.sprites():
            sprite.set_geometry(board_position, board_size[0] / 10)

    def handle_resize(self, board_position, board_size):
        self.set_geometry(board_position, board_size)

        for sprite in self.sprites():
            sprite.set_image()

    def remove_piece(self, coords):
        for sprite in self.sprites():
            if sprite.coords == coords:
                sprite.kill()

    # def handle_resize_end(self):
    #     for sprite in self.sprites():
    #         sprite.handle_resize_end()

    # def clear_square(self, src_bitboard):
    #     list_position = bb_helpers.bitboard_to_index(src_bitboard)
    #     self.square_list[list_position].clear_piece()

    # def update_squares_move(self, src, dest, new_piece_symbol, new_colour, rotation):
    #     self.square_list[src].clear_piece()
    #     self.square_list[dest].clear_piece()
    #     self.square_list[dest].set_piece(piece_symbol=new_piece_symbol, colour=new_colour, rotation=rotation)

    # def update_squares_rotate(self, src, piece_symbol, colour, new_rotation):
    #     self.square_list[src].clear_piece()
    #     self.square_list[src].set_piece(piece_symbol=piece_symbol, colour=colour, rotation=new_rotation)
    # def add_valid_square_overlays(self, valid_bitboard):
    #     if valid_bitboard == EMPTY_BB:
    #         return

    #     list_positions = self.bitboard_to_list_positions(valid_bitboard)
    #     self.valid_square_list_positions = list_positions

    #     for square_position in list_positions:
    #         square = self.square_list[square_position]
    #         square.selected = True

    # def remove_valid_square_overlays(self):
    #     for square_position in self.valid_square_list_positions:
    #         square = self.square_list[square_position]
    #         square.selected = False
    #         square.remove_overlay()

    #     self.valid_square_list_positions = []

    # def draw_valid_square_overlays(self):
    #     for square_position in self.valid_square_list_positions:
    #         square = self.square_list[square_position]
    #         square.draw_overlay()

    # def bitboard_to_list_positions(self, bitboard):
    #     list_positions = []

    #     for square in bb_helpers.occupied_squares(bitboard):
    #         list_positions.append(bb_helpers.bitboard_to_index(square))

    #     return list_positions
```

```
import pygame
from data.assets import GRAPHICS
from data.constants import Colour, Piece
from data.utils.asset_helpers import scale_and_cache
from data.utils.board_helpers import coords_to_screen_pos

class EmptyPiece(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()

        self.image = pygame.Surface((1, 1))
        self.rect = self.image.get_rect()
        self.rect.topleft = (0, 0)

    def set_image(self, type):
        pass

    def set_rect(self):
        pass

    def set_geometry(self, anchor_position, size):
        pass

class PieceSprite(pygame.sprite.Sprite):
    def __init__(self, piece, colour, rotation):
        super().__init__()
        self.colour = colour
        self.rotation = rotation

        self.type = piece
        self.coords = None
        self.size = None

    @property
    def image_name(self):
        return Piece(self.type).name.lower() + '_' + str(self.colour) + '_' + self.rotation

    def set_image(self):
        self.image = scale_and_cache(GRAPHICS[self.image_name], (self.size, self.size))

    def set_geometry(self, new_position, square_size):
        self.size = square_size
        self.rect = pygame.FRect((0, 0, square_size, square_size))

        if self.coords:
            self.rect.topleft = coords_to_screen_pos(self.coords, new_position, square_size)
        else:
            self.rect.topleft = new_position

    def set_coords(self, new_coords):
        self.coords = new_coords
```

```
from data.constants import Piece

FLIP = [
    70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
    60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
    50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
    40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
    6, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    4, 21, 22, 23, 24, 25, 26, 27, 28, 29,
    2, 11, 12, 13, 14, 3, 16, 17, 18, 19,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
]

PSQT = {
    Piece.PYRAMID: [
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ],
    Piece.ANUBIS: [
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    ],
    Piece.SCARAB: [
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
        0, 0, 1, 2, 2, 2, 2, 1, 0, 0,
        0, 0, 1, 2, 3, 3, 2, 1, 0, 0,
        0, 0, 1, 2, 3, 3, 2, 1, 0, 0,
        0, 0, 1, 2, 2, 2, 2, 1, 0, 0,
        0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ],
    Piece.PHAROAH: [
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 2, 2, 2, 2, 0, 0, 0,
        0, 0, 0, 2, 4, 4, 2, 0, 0, 0,
    ],
}
```

```python
from data.states.game.cpu.engines import *
from data.states.game.components.board import Board
from data.constants import Colour, Miscellaneous
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)
# sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b
# scfaRa7/RaRaRaFa6/RaRaRa7/10/10/10/10/9Sa b
# scfa8/10/10/10/10/10/10/8FaSa b

def compare(cls1, cls2, depth, rounds):
    wins = [0, 0]

    board = Board()
    def callback(move):
        board.apply_move(move, add_hash=True)

    cpu1 = cls1(callback=callback, max_depth=depth, verbose='compact')
    cpu2 = cls2(callback=callback, max_depth=depth, verbose='compact')

    for i in range(rounds):
        board = Board(fen_string="scfa8/10/10/10/10/10/10/8FaSa b")
        ply = 0

        if i % 2 == 0:
            players = { Colour.BLUE: cpu1, Colour.RED: cpu2, Miscellaneous.DRAW: 'DRAW' }
        else:
            players = { Colour.BLUE: cpu2, Colour.RED: cpu1, Miscellaneous.DRAW: 'DRAW' }

        while (winner := board.check_win()) is None:
            players[board.get_active_colour()].find_move(board, None)
            ply += 1
            logger.debug('PLY:', ply)

        if winner == Miscellaneous.DRAW:
            wins[0] += 0.5
            wins[1] += 0.5
        else:
            if players[winner] == cpu1:
                wins[0] += 1
            else:
                wins[1] += 1

        logger.debug(f'ROUND {i + 1} | WINNER: {players[winner]} | PLY: {ply}')

    logger.debug(f'{cpu1} SCORE: {wins[0]} | {cpu2} SCORE: {wins[1]}')

compare(TTNegamaxCPU, TTNegamaxCPU, 2, 1)
```

```
import time
from pprint import PrettyPrinter
from data.constants import Colour, Score, Miscellaneous
from data.states.game.cpu.evaluator import Evaluator
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)
printer = PrettyPrinter(indent=2, sort_dicts=False)

class BaseCPU:
    def __init__(self, callback, verbose=True):
        self._evaluator = Evaluator(verbose=False)
        self._verbose = verbose
        self._callback = callback
        self._stats = {}

    def initialise_stats(self):
        self._stats = {
            'nodes': 0,
            'leaf_nodes' : 0,
            'draws': 0,
            'mates': 0,
            'ms_per_node': 0,
            'time_taken': time.time()
        }

    def print_stats(self, score, move):
        """
        Prints statistics after traversing tree.

        Args:
            score (int): Final score obtained after traversal.
            move (Move): Best move obtained after traversal.
        """
        if self._verbose is False:
            return

        self._stats['time_taken'] = round(1000 * (time.time() - self._stats['time_taken']), 3)
        self._stats['ms_per_node'] = round(self._stats['time_taken'] / self._stats['nodes'], 3)

        # Prints stats across multiple lines
        if self._verbose is True:
            logger.info(f'\n\n'
                        f'{self.__str__()} Search Results:\n'
                        f'{printer.pformat(self._stats)}\n'
                        f'Best score:  {score}   Best move: {move}\n'
                        )

        # Prints stats in a compacted format
        elif self._verbose.lower() == 'compact':
```

```python
            logger.info(self._stats)
            logger.info(f'Best score: {score}   Best move: {move}')

    def find_move(self, board, stop_event=None):
        raise NotImplementedError

    def search(self, board, depth, stop_event, absolute=False, **kwargs):
        if stop_event and stop_event.is_set():
            raise Exception(f'Thread killed - stopping minimax function ({self.__str__}.search)')

        self._stats['nodes'] += 1

        if (winner := board.check_win()) is not None:
            self._stats['leaf_nodes'] += 1
            return self.process_win(winner, depth, absolute)

        if depth == 0:
            self._stats['leaf_nodes'] += 1
            return self._evaluator.evaluate(board, absolute), None

    def process_win(self, winner, depth, absolute):
        self._stats['leaf_nodes'] += 1

        if winner == Miscellaneous.DRAW:
            self._stats['draws'] += 1
            return 0, None
        elif winner == Colour.BLUE or absolute:
            self._stats['mates'] += 1
            return Score.CHECKMATE + depth, None
        elif winner == Colour.RED:
            self._stats['mates'] += 1
            return -Score.CHECKMATE - depth, None

    def __str__(self):
        return self.__class__.__name__
```

```python
import threading
import time
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class CPUThread(threading.Thread):
    def __init__(self, cpu, verbose=False):
        super().__init__()
        self._stop_event = threading.Event()
        self._running = True
        self._verbose = verbose
        self.daemon = True

        self._board = None
        self._cpu = cpu

    def kill_thread(self):
        """
        Kills the CPU and terminates the thread by stopping the run loop.
        """
        self.stop_cpu()
        self._running = False

    def stop_cpu(self):
        """
        Kills the CPU's move search.
        """
        self._stop_event.set()
        self._board = None

    def start_cpu(self, board):
        """
        Starts the CPU's move search.

        Args:
            board (Board): The current board state.
        """
        self._stop_event.clear()
        self._board = board

    def run(self):
        """
        Periodically checks if the board variable is set.
        If it is, then starts CPU search.
        """
        while self._running:
            if self._board and self._cpu:
                self._cpu.find_move(self._board, self._stop_event)
                self.stop_cpu()
            else:
                time.sleep(1)
                if self._verbose:
                    logger.debug(f'(CPUThread.run) Thread {threading.get_native_id()} idling...')
```

```
from data.utils.bitboard_helpers import pop_count, occupied_squares, bitboard_to_index
from data.states.game.components.psqt import PSQT, FLIP
from data.managers.logs import initialise_logger
from data.constants import Colour, Piece, Score

logger = initialise_logger(__name__)

class Evaluator:
    def __init__(self, verbose=True):
        self._verbose = verbose

    def evaluate(self, board, absolute=False):
        """
        Evaluates and returns a numerical score for the board state.

        Args:
            board (Board): The current board state.
            absolute (bool): Whether to always return the absolute score from the active colour's perspective (for NegaMax).

        Returns:
            int: Score representing advantage/disadvantage for the player.
        """
        blue_score = (
            self.evaluate_material(board, Colour.BLUE),
            self.evaluate_position(board, Colour.BLUE),
            self.evaluate_mobility(board, Colour.BLUE),
            self.evaluate_pharoah_safety(board, Colour.BLUE)
        )

        red_score = (
            self.evaluate_material(board, Colour.RED),
            self.evaluate_position(board, Colour.RED),
            self.evaluate_mobility(board, Colour.RED),
            self.evaluate_pharoah_safety(board, Colour.RED)
        )

        if self._verbose:
            logger.info(f'Material: {blue_score[0]} | {red_score[0]}')
            logger.info(f'Position: {blue_score[1]} | {red_score[1]}')
            logger.info(f'Mobility: {blue_score[2]} | {red_score[2]}')
            logger.info(f'Safety: {blue_score[3]} | {red_score[3]}')
            logger.info(f'Overall score: {sum(blue_score) - sum(red_score)}')

        if absolute and board.get_active_colour() == Colour.RED:
            return sum(red_score) - sum(blue_score)
        else:
            return sum(blue_score) - sum(red_score)

    def evaluate_material(self, board, colour):
        """
        Evaluates the material score for a given colour.

        Args:
            board (Board): The current board state.
            colour (Colour): The colour to evaluate.

        Returns:
            int: Sum of all piece scores.
        """
        return (
            Score.SPHINX * board.bitboards.get_piece_count(Piece.SPHINX, colour) +
            Score.PYRAMID * board.bitboards.get_piece_count(Piece.PYRAMID, colour)
     +
            Score.ANUBIS * board.bitboards.get_piece_count(Piece.ANUBIS, colour) +
            Score.SCARAB * board.bitboards.get_piece_count(Piece.SCARAB, colour)
        )

    def evaluate_position(self, board, colour):
        """
        Evaluates the positional score for a given colour.

        Args:
            board (Board): The current board state.
            colour (Colour): The colour to evaluate.

        Returns:
            int: Score representing positional advantage/disadvantage.
        """
        score = 0

        for piece in Piece:
            if piece == Piece.SPHINX:
                continue

            piece_bitboard = board.bitboards.get_piece_bitboard(piece, colour)

            for bitboard in occupied_squares(piece_bitboard):
                index = bitboard_to_index(bitboard)
                # Flip PSQT if using from blue player's perspective
                index = FLIP[index] if colour == Colour.BLUE else index

                score += PSQT[piece][index] * Score.POSITION

        return score

    def evaluate_mobility(self, board, colour):
        """
        Evaluates the mobility score for a given colour.

        Args:
            board (Board): The current board state.
            colour (Colour): The colour to evaluate.

        Returns:
            int: Score on numerical representation of mobility.
        """
        number_of_moves = board.get_mobility(colour)
        return number_of_moves * Score.MOVE

    def evaluate_pharoah_safety(self, board, colour):
        """
        Evaluates the safety of the Pharoah for a given colour.

        Args:
            board (Board): The current board state.
            colour (Colour): The colour to evaluate.

        Returns:
            int: Score representing mobility of the Pharoah.
        """
        pharoah_bitboard = board.bitboards.get_piece_bitboard(Piece.PHAROAH, colour)

        if pharoah_bitboard:
            pharoah_available_moves = pop_count(board.get_valid_squares(pharoah_bitboard, colour))
            return (8 - pharoah_available_moves) * Score.PHAROAH_SAFETY
        else:
            return 0
```

```
from data.states.game.cpu.evaluator import Evaluator
from data.constants import Colour
from data.utils.bitboard_helpers import print_bitboard, pop_count

class SimpleEvaluator:
    def __init__(self):
        self._evaluator = Evaluator(verbose=False)
        self._cache = {}

    def evaluate(self, board):
        if (hashed := board.to_hash()) in self._cache:
            return self._cache[hashed]

        score = self._evaluator.evaluate_material(board, board.get_active_colour())
        self._cache[hashed] = score

        return score

class MoveOrderer:
    def __init__(self):
        self._evaluator = SimpleEvaluator()

```

```python
    # def get_eval(self, board, move):
    #     laser_result = board.apply_move(move)
    #     score = self._evaluator.evaluate(board)
    #     board.undo_move(move, laser_result)
    #     return score

    # def score_moves(self, board, moves):
    #     for i in range(len(moves)):
    #         score = self.get_eval(board, moves[i])
    #         moves[i] = (moves[i], score)

    #     return moves

    def best_move_to_front(self, moves, start_idx, hint):
        for i in range(start_idx + 1, len(moves)):
            if moves[i].src in hint:
                moves[i], moves[start_idx] = moves[start_idx], moves[i]
                return

    def get_moves(self, board, hint=None):
        colour = board.get_active_colour()
        moves = list(board.generate_all_moves(colour))

        for i in range(len(moves)):
            if hint:
                self.best_move_to_front(moves, i, hint)

            yield moves[i]
```

```python
from data.constants import Score, Colour
from data.states.game.cpu.transposition_table import TranspositionTable
from data.states.game.cpu.base import BaseCPU
from pprint import pprint

class MinimaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose):
        super().__init__(callback, verbose)
        self._max_depth = max_depth

    def find_move(self, board, stop_event):
        # No bit_length bug as None type returned, so Move __str__ called on NoneType I think (just deal with None being returned)
        try:
            best_move = self.search(board, self._max_depth, -Score.INFINITE, Score.INFINITE, stop_event)

            if self._verbose:
                print('\nCPU Search Results:')
                pprint(self._stats)
                print('Best move:', best_move, '\n')

                self._callback(self._best_move)
        except Exception as error:
            print('(MinimaxBase.find_move) Error has occured:')
            raise error

    def search(self, board, depth, alpha, beta, stop_event):
        if stop_event.is_set():
            raise Exception('Thread killed - stopping minimax function (CPU.minimax)')

        # cached_move, cached_score = self._transposition_table.get_entry(hash_key
=board.bitboards.get_hash(), depth=depth, alpha=alpha, beta=beta)
    # if cached_move or cached_score:
    #     if depth == self._max_depth:
    #         self._best_move = cached_move
    #     return cached_score


    if depth == 0:
        return self.evaluate(board)

    is_maximiser = board.get_active_colour() == Colour.BLUE

    if is_maximiser:
        score = -Score.INFINITE

        for move in board.generate_all_moves(board.get_active_colour()):
            before, before_score = board.bitboards.get_rotation_string(), self
.evaluate(board)

            laser_result = board.apply_move(move)
            new_score = self.minimax(board, depth - 1, alpha, beta, False,
stop_event)

            if new_score >= score:
                score = new_score

                if depth == self._max_depth:
                    self._best_move = move

            board.undo_move(move, laser_result)

            alpha = max(alpha, score)
            if depth == self._max_depth: # https://stackoverflow.com/questions
/31429974/alphabeta-pruning-alpha-equals-or-greater-than-beta-why-equals
                if beta < alpha:
                    break
            else:
                if beta <= alpha:
                    break

            after, after_score = board.bitboards.get_rotation_string(), self.
evaluate(board)
            if (before != after or before_score != after_score):
                print('shit\n\n')

        return score

    else:
        score = Score.INFINITE

        for move in board.generate_all_moves(board.get_active_colour()):
            bef, before_score = board.bitboards.get_rotation_string(), self.
evaluate(board)

            laser_result = board.apply_move(move)
            new_score = self.minimax(board, depth - 1, alpha, beta, False,
stop_event)

            if new_score <= score:
                score = new_score
                if depth == self._max_depth:
                    self._best_move = move

                board.undo_move(move, laser_result)

                beta = min(beta, score)
                if depth == self._max_depth:
                    if beta < alpha:
                        break
                else:
                    if beta <= alpha:
                        break

                after, after_score = board.bitboards.get_rotation_string(), self.evaluate(board)
                if (bef != after or before_score != after_score):
                    print('shit\n\n')
                    raise ValueError

            return score
```

```
from data.constants import TranspositionFlag

class TranspositionEntry:
    def __init__(self, score, move, flag, hash_key, depth):
        self.score = score
        self.move = move
        self.flag = flag
        self.hash_key = hash_key
        self.depth = depth

class TranspositionTable:
    def __init__(self, max_entries=100000):
        self._max_entries = max_entries
        self._table = dict()

    def calculate_entry_index(self, hash_key):
        """
        Gets the dictionary key for a given Zobrist hash.

        Args:
            hash_key (int): A Zobrist hash.

        Returns:
            int: Key for the given hash.
        """
        # return hash_key % self._max_entries
        return hash_key

    def insert_entry(self, score, move, hash_key, depth, alpha, beta):
        """
        Inserts an entry into the transposition table.

        Args:
            score (int): The evaluation score.
            move (Move): The best move found.
            hash_key (int): The Zobrist hash key.
            depth (int): The depth of the search.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.

        Raises:
            Exception: Invalid depth or score.
        """
```

```python
        if depth == 0 or alpha < score < beta:
            flag = TranspositionFlag.EXACT
            score = score
        elif score <= alpha:
            flag = TranspositionFlag.UPPER
            score = alpha
        elif score >= beta:
            flag = TranspositionFlag.LOWER
            score = beta
        else:
            raise Exception('(TranspositionTable.insert_entry)')

        self._table[self.calculate_entry_index(hash_key)] = TranspositionEntry(score, move, flag, hash_key, depth)

        if len(self._table) > self._max_entries:
            # Removes the longest-existing entry to free up space for more up-to-date entries
            # Expression to remove leftmost item taken from https://docs.python.org/3/library/collections.html#ordereddict-objects
            (k := next(iter(self._table)), self._table.pop(k))

    def get_entry(self, hash_key, depth, alpha, beta):
        """
        Gets an entry from the transposition table.

        Args:
            hash_key (int): The Zobrist hash key.
            depth (int): The depth of the search.
            alpha (int): The alpha value for pruning.
            beta (int): The beta value for pruning.

        Returns:
            tuple[int, Move] | tuple[None, None]: The evaluation score and the best move found, if entry exists.
        """
        index = self.calculate_entry_index(hash_key)

        if index not in self._table:
            return None, None

        entry = self._table[index]

        if entry.hash_key == hash_key and entry.depth >= depth:
            if entry.flag == TranspositionFlag.EXACT:
                return entry.score, entry.move

            if entry.flag == TranspositionFlag.LOWER and entry.score >= beta:
                return entry.score, entry.move

            if entry.flag == TranspositionFlag.UPPER and entry.score <= alpha:
                return entry.score, entry.move

        return None, None
```

```python
from random import randint
from data.utils.bitboard_helpers import bitboard_to_index
from data.constants import Piece, Colour, Rotation

# Initialise random values for each piece type on every square
# (5 x 2 colours) pieces + 4 rotations, for 80 squares
zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)]
```

```python
# Hash for when the red player's move
red_move_hash = randint(0, 2 ** 64)

# Maps piece to the correct random value
piece_lookup = {
    Colour.BLUE: {
        piece: i for i, piece in enumerate(Piece)
    },
    Colour.RED: {
        piece: i + 5 for i, piece in enumerate(Piece)
    },
}

# Maps rotation to the correct random value
rotation_lookup = {
    rotation: i + 10 for i, rotation in enumerate(Rotation)
}

class ZobristHasher:
    def __init__(self):
        self.hash = 0

    def get_piece_hash(self, index, piece, colour):
        """
        Gets the random value for the piece type on the given square.

        Args:
            index (int): The index of the square.
            piece (Piece): The piece on the square.
            colour (Colour): The colour of the piece.

        Returns:
            int: A 64-bit value.
        """
        piece_index = piece_lookup[colour][piece]
        return zobrist_table[index][piece_index]

    def get_rotation_hash(self, index, rotation):
        """
        Gets the random value for theon the given square.

        Args:
            index (int): The index of the square.
            rotation (Rotation): The rotation on the square.
            colour (Colour): The colour of the piece.

        Returns:
            int: A 64-bit value.
        """
        rotation_index = rotation_lookup[rotation]
        return zobrist_table[index][rotation_index]

    def apply_piece_hash(self, bitboard, piece, colour):
        """
        Updates the Zobrist hash with a new piece.

        Args:
            bitboard (int): The bitboard representation of the square.
            piece (Piece): The piece on the square.
            colour (Colour): The colour of the piece.
        """
        index = bitboard_to_index(bitboard)
        piece_hash = self.get_piece_hash(index, piece, colour)
        self.hash ^= piece_hash

    def apply_rotation_hash(self, bitboard, rotation):
        """Updates the Zobrist hash with a new rotation.

        Args:
            bitboard (int): The bitboard representation of the square.
            rotation (Rotation): The rotation on the square.
        """
        index = bitboard_to_index(bitboard)
        rotation_hash = self.get_rotation_hash(index, rotation)
        self.hash ^= rotation_hash

    def apply_red_move_hash(self):
        """
        Applies the Zobrist hash for the red player's move.
        """
        self.hash ^= red_move_hash
```

```

from data.states.game.cpu.move_orderer import MoveOrderer
from data.states.game.cpu.base import BaseCPU
from data.constants import Score, Colour
from random import choice
from data.utils.bitboard_helpers import print_bitboard
orderer = MoveOrderer()

class ABMinimaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose=True):
        super().__init__(callback, verbose)
        self._max_depth = max_depth

    def initialise_stats(self):
        """
        Initialises the number of prunes to the statistics dictionary to be logged.
        """
        super().initialise_stats()
        self._stats['beta_prunes'] = 0
        self._stats['alpha_prunes'] = 0

    def find_move(self, board, stop_event):
        """
        Finds the best move for the current board state.

        Args:
            board (Board): The current board state.
            stop_event (threading.Event): Event used to kill search from an external class.
        """
        self.initialise_stats()
        best_score, best_move = self.search(board, self._max_depth, -Score.INFINITE, Score.INFINITE, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, depth, alpha, beta, stop_event, hint=None):
        """
        Recursively DFS through minimax tree while pruning branches using the alpha and beta bounds.

        Args:
            board (Board): The current board state.
            depth (int): The current search depth.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.
            stop_event (threading.Event): Event used to kill search from an external class.

        Returns:
            tuple[int, Move]: The best score and the best move found.
        """
        if (base_case := super().search(board, depth, stop_event)):
            return base_case

        best_move = None

        # Blue is the maximising player
        if board.get_active_colour() == Colour.BLUE:
            max_score = -Score.INFINITE

            for move in orderer.get_moves(board, hint):
                laser_result = board.apply_move(move)
                new_score = self.search(board, depth - 1, alpha, beta, stop_event, laser_result.pieces_on_trajectory)[0]

                if new_score > max_score:
                    max_score = new_score
                    best_move = move

                board.undo_move(move, laser_result)

                alpha = max(alpha, max_score)

                if beta <= alpha:
                    self._stats['alpha_prunes'] += 1
                    break

            return max_score, best_move

        else:
            min_score = Score.INFINITE

            for move in orderer.get_moves(board, hint):
                laser_result = board.apply_move(move)
                new_score = self.search(board, depth - 1, alpha, beta, stop_event, laser_result.pieces_on_trajectory)[0]

                if new_score < min_score:
                    min_score = new_score
                    best_move = move

                board.undo_move(move, laser_result)

                beta = min(beta, min_score)
                if beta <= alpha:
                    self._stats['beta_prunes'] += 1
                    break

            return min_score, best_move
```

```python

class ABNegamaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose=True):
        super().__init__(callback, verbose)
        self._max_depth = max_depth

    def initialise_stats(self):
        """Initialises the statistics for the search."""
        super().initialise_stats()
        self._stats['beta_prunes'] = 0

    def find_move(self, board, stop_event):
        """Finds the best move for the current board state.

        Args:
            board (Board): The current board state.
            stop_event (threading.Event): The event to signal stopping the search.
        """
        self.initialise_stats()
        best_score, best_move = self.search(board, self._max_depth, -Score.INFINITE, Score.INFINITE, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, depth, alpha, beta, stop_event):
        """Searches for the best move using the Alpha-Beta Negamax algorithm.

        Args:
            board (Board): The current board state.
            depth (int): The current depth in the game tree.
            alpha (int): The alpha value for pruning.
            beta (int): The beta value for pruning.
            stop_event (threading.Event): The event to signal stopping the search.

        Returns:
            tuple: The best score and the best move found.
        """
        if (base_case := super().search(board, depth, stop_event, absolute=True)):
            return base_case

        best_move = None
        best_score = alpha

        for move in board.generate_all_moves(board.get_active_colour()):
            laser_result = board.apply_move(move)

            new_score = self.search(board, depth - 1, -beta, -best_score, stop_event)[0]
            new_score = -new_score

            if new_score > best_score:
                best_score = new_score
                best_move = move
            elif new_score == best_score:
                best_move = choice([best_move, move])

            board.undo_move(move, laser_result)

            if best_score >= beta:
                self._stats['beta_prunes'] += 1
                break

        return best_score, best_move
```

```
from data.states.game.cpu.engines.transposition_table import TranspositionTableMixin
from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
from data.constants import Score

class IterativeDeepeningMixin:
    def find_move(self, board, stop_event):
        best_move = None

        for depth in range(1, self._max_depth + 1):
            self.initialise_stats()
            self._stats['ID_depth'] = depth

            best_score, best_move = self.search(board, depth, -Score.INFINITE, Score.INFINITE, stop_event)

            if self._verbose:
                self.print_stats(best_score, best_move)

        self._callback(best_move)

class IDMinimaxCPU(TranspositionTableMixin, IterativeDeepeningMixin, ABMinimaxCPU):
    def initialise_stats(self):
        super().initialise_stats()
        self._stats['cache_hits'] = 0

    def print_stats(self, score, move):
        self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] / self._stats['nodes'], 3)
        self._stats['cache_entries'] = len(self._table._table)
        super().print_stats(score, move)

class IDNegamaxCPU(TranspositionTableMixin, IterativeDeepeningMixin, ABNegamaxCPU):
    def initialise_stats(self):
        super().initialise_stats()
        self._stats['cache_hits'] = 0

    def print_stats(self, score, move):
        self._stats['cache_hits_percentage'] = self._stats['cache_hits'] / self._stats['nodes']
        self._stats['cache_entries'] = len(self._table._table)
        super().print_stats(score, move)
```

```
from data.states.game.cpu.base import BaseCPU
from data.constants import Score, Colour
from random import choice

class MinimaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose=False):
        super().__init__(callback, verbose)
        self._max_depth = max_depth

    def find_move(self, board, stop_event):
        """
        Finds the best move for the current board state.

        Args:
            board (Board): The current board state.
            stop_event (threading.Event): Event used to kill search from an external class.
        """
        self.initialise_stats()
        best_score, best_move = self.search(board, self._max_depth, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, depth, stop_event):
        """
        Recursively DFS through minimax tree with evaluation score.

        Args:
            board (Board): The current board state.
            depth (int): The current search depth.
            stop_event (threading.Event): Event used to kill search from an external class.
        Returns:
            tuple[int, Move]: The best score and the best move found.
        """
        if (base_case := super().search(board, depth, stop_event)):
            return base_case

        best_move = None

        # Blue is the maximising player
        if board.get_active_colour() == Colour.BLUE:
            max_score = -Score.INFINITE

            for move in board.generate_all_moves(Colour.BLUE):
                laser_result = board.apply_move(move)


                new_score = self.search(board, depth - 1, stop_event)[0]

                # if depth < self._max_depth:
                #     print('DEPTH', depth, new_score, move)

                if new_score > max_score:
                    max_score = new_score
                    best_move = move

                    if new_score == (Score.CHECKMATE + self._max_depth):
                        board.undo_move(move, laser_result)
                        return max_score, best_move

                elif new_score == max_score:
                    # If evaluated scores are equal, pick a random move
                    best_move = choice([best_move, move])

                board.undo_move(move, laser_result)

            return max_score, best_move

        else:
            min_score = Score.INFINITE
```

```python

            for move in board.generate_all_moves(Colour.RED):
                laser_result = board.apply_move(move)
                # print('DEPTH', depth, move)
                new_score = self.search(board, depth - 1, stop_event)[0]

                if new_score < min_score:
                    # print('setting new', new_score, move)
                    min_score = new_score
                    best_move = move

                    if new_score == (-Score.CHECKMATE - self._max_depth):
                        board.undo_move(move, laser_result)
                        return min_score, best_move

                elif new_score == min_score:
                    best_move = choice([best_move, move])

                board.undo_move(move, laser_result)

            return min_score, best_move
```

```python
from data.constants import Score, Colour, Miscellaneous, MoveType
from data.states.game.cpu.base import BaseCPU
from data.utils.bitboard_helpers import print_bitboard, is_occupied
from random import choice, randint
from copy import deepcopy

class NegamaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose=False):
        super().__init__(callback, verbose)
        self._max_depth = max_depth

    def find_move(self, board, stop_event):
        self.initialise_stats()
        best_score, best_move = self.search(board, self._max_depth, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, depth, stop_event, moves=None):
        if (base_case := super().search(board, depth, stop_event, absolute=True)):
            return base_case

        best_move = None
        best_score = -Score.INFINITE

        for move in board.generate_all_moves(board.get_active_colour()):
            laser_result = board.apply_move(move)

            new_score = self.search(board, depth - 1, stop_event)[0]
            new_score = -new_score

            if new_score > best_score:
                best_score = new_score
                best_move = move
            elif new_score == best_score:
                best_move = choice([best_move, move])

            board.undo_move(move, laser_result)

        return best_score, best_move
```

```python
from data.states.game.cpu.base import BaseCPU
from data.constants import Colour, Score

class SimpleCPU(BaseCPU):
    def __init__(self, callback, verbose=True):
        super().__init__(callback, verbose)

    def find_move(self, board, stop_event=None):
        self.initialise_stats()
        best_score, best_move = self.search(board, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, stop_event):
        if stop_event and stop_event.is_set():
            raise Exception('Thread killed - stopping simple function (SimpleCPU.search)')

        active_colour = board.bitboards.active_colour
        best_score = -Score.INFINITE if active_colour == Colour.BLUE else Score.INFINITE
        best_move = None

        for move in board.generate_all_moves(active_colour):
            laser_result = board.apply_move(move)

            self._stats['nodes'] += 1

            if winner := board.check_win() is not None:
                self.process_win(winner)
            else:
                self._stats['leaf_nodes'] += 1

            score = self._evaluator.evaluate(board)

            if (active_colour == Colour.BLUE and score > best_score) or (active_colour == Colour.RED and score < best_score):
                best_move = move
                best_score = score

            board.undo_move(move, laser_result)

        return best_score, best_move
```

```python
from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
from data.states.game.cpu.transposition_table import TranspositionTable

class TranspositionTableMixin:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._table = TranspositionTable()

    def find_move(self, *args, **kwargs):
        self._table = TranspositionTable()
        super().find_move(*args, **kwargs)

    def search(self, board, depth, alpha, beta, stop_event, hint=None):
        """
        Searches transposition table for a cached move before running a full search if necessary.
        Caches the searched result.

        Args:
            board (Board): The current board state.
            depth (int): The current search depth.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.
            stop_event (threading.Event): Event used to kill search from an external class.

        Returns:
            tuple[int, Move]: The best score and the best move found.
        """
        hash = board.to_hash()
        score, move = self._table.get_entry(hash, depth, alpha, beta)

        if score is not None:
            self._stats['cache_hits'] += 1
            self._stats['nodes'] += 1

            return score, move
        else:
            # If board hash entry not found in cache, run a full search
            score, move = super().search(board, depth, alpha, beta, stop_event, hint)
            self._table.insert_entry(score, move, hash, depth, alpha, beta)

            return score, move

class TTMinimaxCPU(TranspositionTableMixin, ABMinimaxCPU):
    def initialise_stats(self):
        """
        Initialises cache statistics to be logged.
        """
        super().initialise_stats()
        self._stats['cache_hits'] = 0

    def print_stats(self, score, move):
        """
        Logs the statistics for the search.

        Args:
            score (int): The best score found.
            move (Move): The best move found.
        """
        # Calculate number of cached entries retrieved as a percentage of all nodes
        self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] / self._stats['nodes'], 3)
        self._stats['cache_entries'] = len(self._table._table)
        super().print_stats(score, move)

class TTNegamaxCPU(TranspositionTableMixin, ABNegamaxCPU):
    """Negamax CPU engine with transposition table support."""

    def initialise_stats(self):
        """Initialises the statistics for the search."""
        super().initialise_stats()
        self._stats['cache_hits'] = 0

    def print_stats(self, score, move):
        """Prints the statistics for the search.

        Args:
            score (int): The best score found.
            move (Move): The best move found.
        """
        self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] / self._stats['nodes'], 3)
        self._stats['cache_entries'] = len(self._table._table)
        super().print_stats(score, move)
```

```python
from data.states.game.cpu.engines.simple import SimpleCPU
from data.states.game.cpu.engines.negamax import NegamaxCPU
from data.states.game.cpu.engines.minimax import MinimaxCPU
from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
from data.states.game.cpu.engines.iterative_deepening import IDMinimaxCPU, IDNegamaxCPU
from data.states.game.cpu.engines.transposition_table import TTMinimaxCPU, TTNegamaxCPU
```

```python
import pygame
from data.constants import GameEventType, MoveType, StatusText, Miscellaneous
from data.utils import bitboard_helpers as bb_helpers
from data.states.game.components.move import Move
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class GameController:
    def __init__(self, model, view, win_view, pause_view, to_menu, to_review, to_new_game):
        self._model = model
        self._view = view
        self._win_view = win_view
        self._pause_view = pause_view

        self._to_menu = to_menu
        self._to_review = to_review
        self._to_new_game = to_new_game

        self._view.initialise_timers()
        self._win_view.set_win_type('CAPTURE')

    def cleanup(self, next):
        """
        Handles game quit, either leaving to main menu or restarting a new game.

        Args:
            next (str): New state to switch to.
        """
        self._model.kill_thread()

        if next == 'menu':
            self._to_menu()
        elif next == 'game':
            self._to_new_game()
        elif next == 'review':
            self._to_review()

    def make_move(self, move):
        """
        Handles player move.

        Args:
            move (Move): Move to make.
        """
        self._model.make_move(move)
        self._view.set_overlay_coords([], None)

        if self._model.states['CPU_ENABLED']:
            self._model.make_cpu_move()

    def handle_pause_event(self, event):
        """
        Processes events when game is paused.

        Args:
            event (GameEventType): Event to process.

        Raises:
            Exception: If event type is unrecognised.
        """
        game_event = self._pause_view.convert_mouse_pos(event)

        if game_event is None:
            return

        match game_event.type:
            case GameEventType.PAUSE_CLICK:
                self._model.toggle_paused()

            case GameEventType.MENU_CLICK:
                self.cleanup('menu')

            case _:
                raise Exception('Unhandled event type (GameController.handle_event)')

    def handle_winner_event(self, event):
        """
        Processes events when game is over.

        Args:
            event (GameEventType): Event to process.

        Raises:
            Exception: If event type is unrecognised.
        """
        game_event = self._win_view.convert_mouse_pos(event)

        if game_event is None:
            return

        match game_event.type:
            case GameEventType.MENU_CLICK:
                self.cleanup('menu')
                return

            case GameEventType.GAME_CLICK:
                self.cleanup('game')
                return
```

```python

            case GameEventType.REVIEW_CLICK:
                self.cleanup('review')

            case _:
                raise Exception('Unhandled event type (GameController.handle_event)')

    def handle_game_widget_event(self, event):
        """
        Processes events for game GUI widgets.

        Args:
            event (GameEventType): Event to process.

        Raises:
            Exception: If event type is unrecognised.

        Returns:
            CustomEvent | None: A widget event.
        """
        widget_event = self._view.process_widget_event(event)

        if widget_event is None:
            return None

        match widget_event.type:
            case GameEventType.ROTATE_PIECE:
                src_coords = self._view.get_selected_coords()

                if src_coords is None:
                    logger.info('None square selected')
                    return

                move = Move.instance_from_coords(MoveType.ROTATE, src_coords, src_coords, rotation_direction=widget_event.rotation_direction)
                self.make_move(move)

            case GameEventType.RESIGN_CLICK:
                self._model.set_winner(self._model.states['ACTIVE_COLOUR'].get_flipped_colour())
                self._view.handle_game_end(play_sfx=False)
                self._win_view.set_win_type('RESIGN')

            case GameEventType.DRAW_CLICK:
                self._model.set_winner(Miscellaneous.DRAW)
                self._view.handle_game_end(play_sfx=False)
                self._win_view.set_win_type('DRAW')

            case GameEventType.TIMER_END:
                if self._model.states['TIME_ENABLED']:
                    self._model.set_winner(widget_event.active_colour.get_flipped_colour())
                    self._win_view.set_win_type('TIME')
                    self._view.handle_game_end(play_sfx=False)

            case GameEventType.MENU_CLICK:
                self.cleanup('menu')

            case GameEventType.HELP_CLICK:
                self._view.add_help_screen()

            case GameEventType.TUTORIAL_CLICK:
                self._view.add_tutorial_screen()

            case _:
                raise Exception('Unhandled event type (GameController.handle_event)')

        return widget_event.type

    def check_cpu(self):
        """
        Checks if CPU calculations are finished every frame.
        """
        if self._model.states['CPU_ENABLED'] and self._model.states['AWAITING_CPU'] is False:
            self._model.check_cpu()

    def handle_game_event(self, event):
        """
        Processes Pygame events for main game.

        Args:
            event (pygame.Event): If event type is unrecognised.

        Raises:
            Exception: If event type is unrecognised.
        """
        # Pass event for widgets to process
        widget_event = self.handle_game_widget_event(event)

        if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            if event.type != pygame.KEYDOWN:
                game_event = self._view.convert_mouse_pos(event)
            else:
                game_event = None

            if game_event is None:
                if widget_event is None:
                    if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
                        # If user releases mouse click not on a widget
                        self._view.remove_help_screen()
                        self._view.remove_tutorial_screen()
                    if event.type == pygame.MOUSEBUTTONUP:
                        # If user releases mouse click on neither a widget or board
                        self._view.set_overlay_coords(None, None)

                return

            match game_event.type:
                case GameEventType.BOARD_CLICK:
                    if self._model.states['AWAITING_CPU']:
                        return

                    clicked_coords = game_event.coords
                    clicked_bitboard = bb_helpers.coords_to_bitboard(clicked_coords)
                    selected_coords = self._view.get_selected_coords()

                    if selected_coords:
                        if clicked_coords == selected_coords:
```

```
                        # If clicking on an already selected square, start dragging piece on that square
                        self._view.set_dragged_piece(*self._model.get_piece_info(clicked_bitboard))
                        return

                    selected_bitboard = bb_helpers.coords_to_bitboard(selected_coords)
                    available_bitboard = self._model.get_available_moves(selected_bitboard)

                    if bb_helpers.is_occupied(clicked_bitboard, available_bitboard):
                        # If the newly clicked square is not the same as the old one, and is an empty surrounding square, make a move
                        move = Move.instance_from_coords(MoveType.MOVE, selected_coords, clicked_coords)
                        self.make_move(move)
                    else:
                        # If the newly clicked square is not the same as the old one, but is an invalid square, unselect the currently selected square
                        self._view.set_overlay_coords(None, None)

                # Select hovered square if it is same as active colour
                elif self._model.is_selectable(clicked_bitboard):
                    available_bitboard = self._model.get_available_moves(clicked_bitboard)
                    self._view.set_overlay_coords(bb_helpers.bitboard_to_coords_list(available_bitboard), clicked_coords)
                    self._view.set_dragged_piece(*self._model.get_piece_info(clicked_bitboard))

            case GameEventType.PIECE_DROP:
                hovered_coords = game_event.coords

                # if piece is dropped onto the board
                if hovered_coords:
                    hovered_bitboard = bb_helpers.coords_to_bitboard(hovered_coords)
                    selected_coords = self._view.get_selected_coords()
                    selected_bitboard = bb_helpers.coords_to_bitboard(selected_coords)
                    available_bitboard = self._model.get_available_moves(selected_bitboard)

                    if bb_helpers.is_occupied(hovered_bitboard, available_bitboard):
                        # Make a move if mouse is hovered over an empty surrounding square
                        move = Move.instance_from_coords(MoveType.MOVE, selected_coords, hovered_coords)
                        self.make_move(move)

                if game_event.remove_overlay:
                    self._view.set_overlay_coords(None, None)

                self._view.remove_dragged_piece()

            case _:
                raise Exception('Unhandled event type (GameController.handle_event)', game_event.type)

```

```
    def handle_event(self, event):
        """
        Passe a Pygame event to the correct handling function according to the game state.

        Args:
            event (pygame.Event): Event to process.
        """
        if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.MOUSEMOTION, pygame.KEYDOWN]:
            if self._model.states['PAUSED']:
                self.handle_pause_event(event)
            elif self._model.states['WINNER'] is not None:
                self.handle_winner_event(event)
            else:
                self.handle_game_event(event)

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                self._model.toggle_paused()
            elif event.key == pygame.K_l:
                logger.info('\nSTOPPING CPU')
                self._model._cpu_thread.stop_cpu() #temp
```

```
from data.states.game.components.fen_parser import encode_fen_string
from data.constants import Colour, GameEventType, EMPTY_BB
from data.states.game.widget_dict import GAME_WIDGETS
from data.states.game.cpu.cpu_thread import CPUThread
from data.states.game.cpu.engines import ABMinimaxCPU
from data.components.custom_event import CustomEvent
from data.utils.bitboard_helpers import is_occupied
from data.states.game.components.board import Board
from data.utils import input_helpers as ip_helpers
from data.states.game.components.move import Move
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class GameModel:
    def __init__(self, game_config):
        self._listeners = {
            'game': [],
            'win': [],
            'pause': [],
        }
        self._board = Board(fen_string=game_config['FEN_STRING'])

        self.states = {
            'CPU_ENABLED': game_config['CPU_ENABLED'],
            'CPU_DEPTH': game_config['CPU_DEPTH'],
            'AWAITING_CPU': False,
            'WINNER': None,
            'PAUSED': False,
            'ACTIVE_COLOUR': game_config['COLOUR'],
            'TIME_ENABLED': game_config['TIME_ENABLED'],
            'TIME': game_config['TIME'],
            'START_FEN_STRING': game_config['FEN_STRING'],
            'MOVES': [],
            'ZOBRIST_KEYS': []
        }

        self._cpu = ABMinimaxCPU(self.states['CPU_DEPTH'], self.cpu_callback,
verbose=False)
    self._cpu_thread = CPUThread(self._cpu)
    self._cpu_thread.start()
    self._cpu_move = None

    logger.info(f'Initialising CPU depth of {self.states['CPU_DEPTH']}')

def register_listener(self, listener, parent_class):
    """
    Registers listener method of another MVC class.

    Args:
        listener (callable): Listener callback function.
        parent_class (str): Class name.
    """
    self._listeners[parent_class].append(listener)

def alert_listeners(self, event):
    """
    Alerts all registered classes of an event by calling their listener
function.

    Args:
        event (GameEventType): Event to pass as argument.

    Raises:
        Exception: If an unrecgonised event tries to be passed onto listeners.
    """
    for parent_class, listeners in self._listeners.items():
        match event.type:
            case GameEventType.UPDATE_PIECES:
                if parent_class in 'game':
                    for listener in listeners: listener(event)

            case GameEventType.SET_LASER:
                if parent_class == 'game':
                    for listener in listeners: listener(event)

            case GameEventType.PAUSE_CLICK:
                if parent_class in ['pause', 'game']:
                    for listener in listeners:
                        listener(event)

            case _:
                raise Exception('Unhandled event type (GameModel.
alert_listeners)')

def set_winner(self, colour=None):
    """
    Sets winner.

    Args:
        colour (Colour, optional): Describes winnner colour, or draw. Defaults
 to None.
    """
    self.states['WINNER'] = colour

def toggle_paused(self):
    """
    Toggles pause screen, and alerts pause view.
    """
    self.states['PAUSED'] = not self.states['PAUSED']
```

```python
        game_event = CustomEvent.create_event(GameEventType.PAUSE_CLICK)
        self.alert_listeners(game_event)

    def get_terminal_move(self):
        """
        Debugging method for inputting a move from the terminal.

        Returns:
            Move: Parsed move.
        """
        while True:
            try:
                move_type = ip_helpers.parse_move_type(input('Input move type (m/r): '))
                src_square = ip_helpers.parse_notation(input("From: "))
                dest_square = ip_helpers.parse_notation(input("To: "))
                rotation = ip_helpers.parse_rotation(input("Enter rotation (a/b/c/d): "))
                return Move.instance_from_notation(move_type, src_square, dest_square, rotation)
            except ValueError as error:
                logger.warning('Input error (Board.get_move): ' + str(error))

    def make_move(self, move):
        """
        Takes a Move object and applies it to the board.

        Args:
            move (Move): Move to apply.
        """
        colour = self._board.bitboards.get_colour_on(move.src)
        piece = self._board.bitboards.get_piece_on(move.src, colour)
        # Apply move and get results of laser trajectory
        laser_result = self._board.apply_move(move, add_hash=True)

        self.alert_listeners(CustomEvent.create_event(GameEventType.SET_LASER, laser_result=laser_result))

        # Sets new active colour and checks for a win
        self.states['ACTIVE_COLOUR'] = self._board.get_active_colour()
        self.set_winner(self._board.check_win())

        move_notation = move.to_notation(colour, piece, laser_result.hit_square_bitboard)

        self.alert_listeners(CustomEvent.create_event(GameEventType.UPDATE_PIECES, move_notation=move_notation))

        # Adds move to move history list for review screen
        self.states['MOVES'].append({
            'time': {
                Colour.BLUE: GAME_WIDGETS['blue_timer'].get_time(),
                Colour.RED: GAME_WIDGETS['red_timer'].get_time()
            },
            'move': move_notation,
            'laserResult': laser_result
        })

    def make_cpu_move(self):
        """
        Starts CPU calculations on the separate thread.
        """
        self.states['AWAITING_CPU'] = True
        self._cpu_thread.start_cpu(self.get_board())

    def cpu_callback(self, move):
        """
        Callback function passed to CPU thread. Called when CPU stops processing.

        Args:
            move (Move): Move that CPU found.
        """
        if self.states['WINNER'] is None:
            # CPU move passed back to main threadby reassigning variable
            self._cpu_move = move
            self.states['AWAITING_CPU'] = False

    def check_cpu(self):
        """
        Constantly checks if CPU calculations are finished, so that make_move can be run on the main thread.
        """
        if self._cpu_move is not None:
            self.make_move(self._cpu_move)
            self._cpu_move = None

    def kill_thread(self):
        """
        Interrupt and kill CPU thread.
        """
        self._cpu_thread.kill_thread()
        self.states['AWAITING_CPU'] = False

    def is_selectable(self, bitboard):
        """
        Checks if square is occupied by a piece of the current active colour.

        Args:
            bitboard (int): Bitboard representing single square.

        Returns:
            bool: True if square is occupied by a piece of the current active colour. False if not.
        """
        return is_occupied(self._board.bitboards.combined_colour_bitboards[self.states['ACTIVE_COLOUR']], bitboard)

    def get_available_moves(self, bitboard):
        """
        Gets all surrounding empty squares. Used for drawing overlay.

        Args:
            bitboard (int): Bitboard representing single center square.

        Returns:
            int: Bitboard representing all empty surrounding squares.
        """
        if (bitboard & self._board.get_all_active_pieces()) != EMPTY_BB:
            return self._board.get_valid_squares(bitboard)

        return EMPTY_BB

    def get_piece_list(self):
        """
        Returns:
            list[Piece, ...]: Array of all pieces on the board.
        """
        return self._board.get_piece_list()

    def get_piece_info(self, bitboard):
        """
        Args:
            bitboard (int): Square containing piece.

        Returns:
            tuple[Colour, Rotation, Piece]: Piece information.
        """
        colour = self._board.bitboards.get_colour_on(bitboard)
        rotation = self._board.bitboards.get_rotation_on(bitboard)
        piece = self._board.bitboards.get_piece_on(bitboard, colour)
        return (piece, colour, rotation)

    def get_fen_string(self):
        return encode_fen_string(self._board.bitboards)

    def get_board(self):
        return self._board
```

```
import pygame
from data.constants import GameEventType, Colour, StatusText, Miscellaneous, ShaderType
from data.states.game.components.overlay_draw import OverlayDraw
from data.states.game.components.capture_draw import CaptureDraw
from data.states.game.components.piece_group import PieceGroup
from data.states.game.components.laser_draw import LaserDraw
from data.states.game.components.father import DragAndDrop
from data.utils.bitboard_helpers import bitboard_to_coords
from data.utils.board_helpers import screen_pos_to_coords
from data.states.game.widget_dict import GAME_WIDGETS
from data.components.custom_event import CustomEvent
from data.components.widget_group import WidgetGroup
from data.managers.window import window
from data.managers.audio import audio
from data.assets import SFX

class GameView:
    def __init__(self, model):
        self._model = model
        self._hide_pieces = False
        self._selected_coords = None
        self._event_to_func_map = {
            GameEventType.UPDATE_PIECES: self.handle_update_pieces,
            GameEventType.SET_LASER: self.handle_set_laser,
            GameEventType.PAUSE_CLICK: self.handle_pause,
        }

        # Register model event handling with process_model_event()
        self._model.register_listener(self.process_model_event, 'game')

        # Initialise WidgetGroup with map of widgets
        self._widget_group = WidgetGroup(GAME_WIDGETS)
        self._widget_group.handle_resize(window.size)
        self.initialise_widgets()

        self._laser_draw = LaserDraw(self.board_position, self.board_size)
        self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
```

```python
        self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
        self._capture_draw = CaptureDraw(self.board_position, self.board_size)
        self._piece_group = PieceGroup()
        self.handle_update_pieces()

        self.set_status_text(StatusText.PLAYER_MOVE)

    @property
    def board_position(self):
        return GAME_WIDGETS['chessboard'].position

    @property
    def board_size(self):
        return GAME_WIDGETS['chessboard'].size

    @property
    def square_size(self):
        return self.board_size[0] / 10

    def initialise_widgets(self):
        """
        Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
        """
        GAME_WIDGETS['move_list'].reset_move_list()
        GAME_WIDGETS['move_list'].kill()
        GAME_WIDGETS['help'].kill()
        GAME_WIDGETS['tutorial'].kill()

        GAME_WIDGETS['scroll_area'].set_image()

        GAME_WIDGETS['chessboard'].refresh_board()

        GAME_WIDGETS['blue_piece_display'].reset_piece_list()
        GAME_WIDGETS['red_piece_display'].reset_piece_list()

    def set_status_text(self, status):
        """
        Sets text on status text widget.

        Args:
            status (StatusText): The game stage for which text should be displayed for.
        """
        match status:
            case StatusText.PLAYER_MOVE:
                GAME_WIDGETS['status_text'].set_text(f"{self._model.states['ACTIVE_COLOUR'].name}'s turn to move")
            case StatusText.CPU_MOVE:
                GAME_WIDGETS['status_text'].set_text(f"CPU calculating a crazy move...")
            case StatusText.WIN:
                if self._model.states['WINNER'] == Miscellaneous.DRAW:
                    GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring...")
                else:
                    GAME_WIDGETS['status_text'].set_text(f"{self._model.states['WINNER'].name} won!")
            case StatusText.DRAW:
                GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring...")

    def handle_resize(self):
        """
        Handles resizing of the window.
        """
        self._overlay_draw.handle_resize(self.board_position, self.board_size)
        self._capture_draw.handle_resize(self.board_position, self.board_size)
        self._piece_group.handle_resize(self.board_position, self.board_size)
        self._laser_draw.handle_resize(self.board_position, self.board_size)
        self._laser_draw.handle_resize(self.board_position, self.board_size)
        self._widget_group.handle_resize(window.size)

        if self._laser_draw.firing:
            self.update_laser_mask()

    def handle_update_pieces(self, event=None):
        """
        Callback function to update pieces after move.

        Args:
            event (GameEventType, optional): If updating pieces after player move, event contains move information. Defaults to None.
            toggle_timers (bool, optional): Toggle timers on and off for new active colour. Defaults to True.
        """
        piece_list = self._model.get_piece_list()
        self._piece_group.initialise_pieces(piece_list, self.board_position, self.board_size)

        if event:
            GAME_WIDGETS['move_list'].append_to_move_list(event.move_notation)
            GAME_WIDGETS['scroll_area'].set_image()
            audio.play_sfx(SFX['piece_move'])

        if self._model.states['ACTIVE_COLOUR'] == Colour.BLUE:
            self.set_status_text(StatusText.PLAYER_MOVE)
        elif self._model.states['CPU_ENABLED'] is False:
            self.set_status_text(StatusText.PLAYER_MOVE)
        else:
            self.set_status_text(StatusText.CPU_MOVE)

        if self._model.states['TIME_ENABLED']:
            self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
            self.toggle_timer(self._model.states['ACTIVE_COLOUR'].get_flipped_colour(), False)

        if self._model.states['WINNER'] is not None:
            self.handle_game_end()

    def handle_game_end(self, play_sfx=True):
        self.toggle_timer(self._model.states['ACTIVE_COLOUR'], False)
        self.toggle_timer(self._model.states['ACTIVE_COLOUR'].get_flipped_colour(), False)

        if self._model.states['WINNER'] == Miscellaneous.DRAW:
            self.set_status_text(StatusText.DRAW)
        else:
            self.set_status_text(StatusText.WIN)

        if play_sfx:
            audio.play_sfx(SFX['sphinx_destroy_1'])
            audio.play_sfx(SFX['sphinx_destroy_2'])
            audio.play_sfx(SFX['sphinx_destroy_3'])

    def handle_set_laser(self, event):
```

```python
        """
        Callback function to draw laser after move.

        Args:
            event (GameEventType): Contains laser trajectory information.
        """
        laser_result = event.laser_result

        # If laser has hit a piece
        if laser_result.hit_square_bitboard:
            coords_to_remove = bitboard_to_coords(laser_result.hit_square_bitboard)
            self._piece_group.remove_piece(coords_to_remove)

            if laser_result.piece_colour == Colour.BLUE:
                GAME_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit)
            elif laser_result.piece_colour == Colour.RED:
                GAME_WIDGETS['blue_piece_display'].add_piece(laser_result.piece_hit)

            # Draw piece capture GFX
            self._capture_draw.add_capture(
                laser_result.piece_hit,
                laser_result.piece_colour,
                laser_result.piece_rotation,
                coords_to_remove,
                laser_result.laser_path[0][0],
                self._model.states['ACTIVE_COLOUR']
            )

        self._laser_draw.add_laser(laser_result, self._model.states['ACTIVE_COLOUR'])
        self.update_laser_mask()

    def handle_pause(self, event=None):
        """
        Callback function for pausing timer.

        Args:
            event (None): Event argument not used.
        """
        is_active = not(self._model.states['PAUSED'])
        self.toggle_timer(self._model.states['ACTIVE_COLOUR'], is_active)

    def initialise_timers(self):
        """
        Initialises both timers with the correct amount of time and starts the timer for the active colour.
        """
        if self._model.states['TIME_ENABLED']:
            GAME_WIDGETS['blue_timer'].set_time(self._model.states['TIME'] * 60 * 1000)
            GAME_WIDGETS['red_timer'].set_time(self._model.states['TIME'] * 60 * 1000)
        else:
            GAME_WIDGETS['blue_timer'].kill()
            GAME_WIDGETS['red_timer'].kill()

        self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)

    def toggle_timer(self, colour, is_active):
        """
        Stops or resumes timer.

        Args:
            colour (Colour): Timer to toggle.
            is_active (bool): Whether to pause or resume timer.
        """
        if colour == Colour.BLUE:
            GAME_WIDGETS['blue_timer'].set_active(is_active)
        elif colour == Colour.RED:
            GAME_WIDGETS['red_timer'].set_active(is_active)

    def update_laser_mask(self):
        """
        Uses pygame.mask to create a mask for the pieces.
        Used for occluding the ray shader.
        """
        temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
        self._piece_group.draw(temp_surface)
        mask = pygame.mask.from_surface(temp_surface, threshold=127)
        mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255, 0, 0, 255))

        window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)

    def draw(self):
        """
        Draws GUI and pieces onto the screen.
        """
        self._widget_group.update()
        self._capture_draw.update()

        self._widget_group.draw()
        self._overlay_draw.draw(window.screen)

        if self._hide_pieces is False:
            self._piece_group.draw(window.screen)

        self._laser_draw.draw(window.screen)
        self._drag_and_drop.draw(window.screen)
        self._capture_draw.draw(window.screen)

    def process_model_event(self, event):
        """
        Registered listener function for handling GameModel events.
        Each event is mapped to a callback function, and the appropiate one is run.

        Args:
            event (GameEventType): Game event to process.

        Raises:
            KeyError: If an unrecgonised event type is passed as the argument.
        """
        try:
            self._event_to_func_map.get(event.type)(event)
        except:
            raise KeyError('Event type not recognized in Game View (GameView.process_model_event):', event.type)

    def set_overlay_coords(self, available_coords_list, selected_coord):
        """
        Set board coordinates for potential moves overlay.

        Args:
            available_coords_list (list[tuple[int, int]], ...): Array of coordinates
            selected_coord (list[int, int]): Coordinates of selected piece.
        """
        self._selected_coords = selected_coord
        self._overlay_draw.set_selected_coords(selected_coord)
        self._overlay_draw.set_available_coords(available_coords_list)

    def get_selected_coords(self):
        return self._selected_coords

    def set_dragged_piece(self, piece, colour, rotation):
        """
        Passes information of the dragged piece to the dragging drawing class.

        Args:
            piece (Piece): Piece type of dragged piece.
            colour (Colour): Colour of dragged piece.
            rotation (Rotation): Rotation of dragged piece.
        """
        self._drag_and_drop.set_dragged_piece(piece, colour, rotation)

    def remove_dragged_piece(self):
        """
        Stops drawing dragged piece when user lets go of piece.
        """
        self._drag_and_drop.remove_dragged_piece()

    def convert_mouse_pos(self, event):
        """
        Passes information of what mouse cursor is interacting with to a GameController object.

        Args:
            event (pygame.Event): Mouse event to process.

        Returns:
            CustomEvent | None: Contains information what mouse is doing.
        """
        clicked_coords = screen_pos_to_coords(event.pos, self.board_position, self.board_size)

        if event.type == pygame.MOUSEBUTTONDOWN:
            if clicked_coords:
                return CustomEvent.create_event(GameEventType.BOARD_CLICK, coords=clicked_coords)

            else:
                return None

        elif event.type == pygame.MOUSEBUTTONUP:
            if self._drag_and_drop.dragged_sprite:
                piece, colour, rotation = self._drag_and_drop.get_dragged_info()
                piece_dragged = self._drag_and_drop.remove_dragged_piece()
                return CustomEvent.create_event(GameEventType.PIECE_DROP, coords=clicked_coords, piece=piece, colour=colour, rotation=rotation, remove_overlay=piece_dragged)

    def add_help_screen(self):
```

```python
        """
        Draw help overlay when player clicks on the help button.
        """
        self._widget_group.add(GAME_WIDGETS['help'])
        self._widget_group.handle_resize(window.size)

    def add_tutorial_screen(self):
        """
        Draw tutorial overlay when player clicks on the tutorial button.
        """
        self._widget_group.add(GAME_WIDGETS['tutorial'])
        self._widget_group.handle_resize(window.size)
        self._hide_pieces = True

    def remove_help_screen(self):
        GAME_WIDGETS['help'].kill()

    def remove_tutorial_screen(self):
        GAME_WIDGETS['tutorial'].kill()
        self._hide_pieces = False

    def process_widget_event(self, event):
        """
        Passes Pygame event to WidgetGroup to allow individual widgets to process events.

        Args:
            event (pygame.Event): Event to process.

        Returns:
            CustomEvent | None: A widget event.
        """
        return self._widget_group.process_event(event)
```

```python
import pygame
from data.states.game.widget_dict import PAUSE_WIDGETS
from data.constants import GameEventType, PAUSE_COLOUR
from data.components.widget_group import WidgetGroup
from data.managers.window import window
from data.managers.audio import audio

class PauseView:
    def __init__(self, model):
        self._model = model

        self._screen_overlay = pygame.Surface(window.size, pygame.SRCALPHA)
        self._screen_overlay.fill(PAUSE_COLOUR)

        self._widget_group = WidgetGroup(PAUSE_WIDGETS)
        self._widget_group.handle_resize(window.size)

        self._model.register_listener(self.process_model_event, 'pause')

        self._event_to_func_map = {
            GameEventType.PAUSE_CLICK: self.handle_pause_click
        }

        self.states = {
            'PAUSED': False
        }

    def handle_pause_click(self, event):
        self.states['PAUSED'] = not self.states['PAUSED']

        if self.states['PAUSED']:
            audio.pause_sfx()
        else:
            audio.unpause_sfx()

    def handle_resize(self):
        self._screen_overlay = pygame.Surface(window.size, pygame.SRCALPHA)
        self._screen_overlay.fill(PAUSE_COLOUR)
        self._widget_group.handle_resize(window.size)

    def draw(self):
        if self.states['PAUSED']:
            window.screen.blit(self._screen_overlay, (0, 0))
            self._widget_group.draw()

    def process_model_event(self, event):
        try:
            self._event_to_func_map.get(event.type)(event)
        except:
            raise KeyError('Event type not recognized in Paused View (PauseView.process_model_event)', event)

    def convert_mouse_pos(self, event):
        return self._widget_group.process_event(event)
```

```python
from data.constants import Colour, Miscellaneous, CursorMode
from data.components.widget_group import WidgetGroup
from data.states.game.widget_dict import WIN_WIDGETS
from data.managers.window import window
from data.managers.cursor import cursor

class WinView:
    def __init__(self, model):
        self._model = model

        self._widget_group = WidgetGroup(WIN_WIDGETS)
        self._widget_group.handle_resize(window.size)

    def handle_resize(self):
        self._widget_group.handle_resize(window.size)

    def draw(self):
        if self._model.states['WINNER'] is not None:
            if cursor.get_mode() != CursorMode.ARROW:
                cursor.set_mode(CursorMode.ARROW)

            if self._model.states['WINNER'] == Colour.BLUE:
                WIN_WIDGETS['red_won'].kill()
                WIN_WIDGETS['draw_won'].kill()
            elif self._model.states['WINNER'] == Colour.RED:
                WIN_WIDGETS['blue_won'].kill()
                WIN_WIDGETS['draw_won'].kill()
            elif self._model.states['WINNER'] == Miscellaneous.DRAW:
                WIN_WIDGETS['red_won'].kill()
                WIN_WIDGETS['blue_won'].kill()

            self._widget_group.draw()

    def set_win_type(self, win_type):
        WIN_WIDGETS['by_draw'].kill()
        WIN_WIDGETS['by_timeout'].kill()
        WIN_WIDGETS['by_resignation'].kill()
        WIN_WIDGETS['by_checkmate'].kill()

        match win_type:
            case 'CAPTURE':
                self._widget_group.add(WIN_WIDGETS['by_checkmate'])
            case 'DRAW':
                self._widget_group.add(WIN_WIDGETS['by_draw'])
            case 'RESIGN':
                self._widget_group.add(WIN_WIDGETS['by_resignation'])
            case 'TIME':
                self._widget_group.add(WIN_WIDGETS['by_timeout'])

    def convert_mouse_pos(self, event):
        return self._widget_group.process_event(event)
```

```python
import pygame
import sys
from random import randint
from data.utils.asset_helpers import get_rotational_angle
from data.states.menu.widget_dict import MENU_WIDGETS
from data.constants import MenuEventType, ShaderType
from data.utils.asset_helpers import scale_and_cache
from data.managers.logs import initialise_logger
from data.managers.animation import animation
from data.assets import GRAPHICS, MUSIC, SFX
from data.managers.window import window
from data.managers.audio import audio
from data.control import _State

logger = initialise_logger(__file__)

class Menu(_State):
    def __init__(self):
        super().__init__()
        self._fire_laser = False
        self._bloom_mask = None
        self._laser_mask = None

    def cleanup(self):
        super().cleanup()

        window.clear_apply_arguments(ShaderType.BLOOM)
        window.clear_apply_arguments(ShaderType.SHAKE)
        window.clear_effect(ShaderType.CHROMATIC_ABBREVIATION)

        return None

    def startup(self, persist=None):
        super().startup(MENU_WIDGETS, music=MUSIC[f'menu_{randint(1, 3)}'])
        window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.BACKGROUND_BALATRO)
        window.set_effect(ShaderType.CHROMATIC_ABBREVIATION)

        MENU_WIDGETS['credits'].kill()

        self._fire_laser = False
        self._bloom_mask = None
        self._laser_mask = None

        self.draw()
        self.update_masks()

    @property
    def sphinx_center(self):
        return (window.size[0] - self.sphinx_size[0] / 2, window.size[1] - self.sphinx_size[1] / 2)

    @property
    def sphinx_size(self):
        return (min(window.size) * 0.1, min(window.size) * 0.1)

    @property
    def sphinx_rotation(self):
        mouse_pos = (pygame.mouse.get_pos()[0], pygame.mouse.get_pos()[1] + 0.01)
        return -get_rotational_angle(mouse_pos, self.sphinx_center)

    def get_event(self, event):
        if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            MENU_WIDGETS['credits'].kill()

        if event.type == pygame.MOUSEBUTTONDOWN:
            self._fire_laser = True
            audio.play_sfx(SFX['menu_laser_windup'])
            audio.play_sfx(SFX['menu_laser_loop'], loop=True)
            animation.set_timer(SFX['menu_laser_loop'].get_length() * 1000 / 2, lambda: audio.play_sfx(SFX['menu_laser_loop'], loop=True) if self._fire_laser else ...) # OVERLAP TWO LOOPS TO HIDE TRANSITION

        elif event.type == pygame.MOUSEBUTTONUP:
            self._fire_laser = False

            window.clear_effect(ShaderType.RAYS)
            animation.set_timer(300, lambda: window.clear_effect(ShaderType.SHAKE))
            audio.stop_sfx(1000)

        widget_event = self._widget_group.process_event(event)

        if widget_event is None:
            return

        match widget_event.type:
            case None:
                return

            case MenuEventType.CONFIG_CLICK:
                self.next = 'config'
                self.done = True
            case MenuEventType.SETTINGS_CLICK:
                self.next = 'settings'
                self.done = True
            case MenuEventType.BROWSER_CLICK:
                self.next = 'browser'
                self.done = True
            case MenuEventType.QUIT_CLICK:
                pygame.quit()
                sys.exit()
                logger.info('quitting...')
            case MenuEventType.CREDITS_CLICK:
                self._widget_group.add(MENU_WIDGETS['credits'])

    def draw_sphinx(self):
        sphinx_surface = scale_and_cache(GRAPHICS['sphinx_0_b'], self.sphinx_size)
        sphinx_surface = pygame.transform.rotate(sphinx_surface, self.sphinx_rotation)
        sphinx_rect = pygame.FRect(0, 0, *self.sphinx_size)
        sphinx_rect.center = self.sphinx_center

        window.screen.blit(sphinx_surface, sphinx_rect)

    def update_masks(self):
        self.draw()

        widget_mask = window.screen.copy()
        laser_mask = pygame.mask.from_surface(widget_mask)
        laser_mask = laser_mask.to_surface(setcolor=(255, 0, 0, 255), unsetcolor=(0, 0, 0, 255))
        pygame.draw.rect(laser_mask, (0, 0, 0), (window.screen.width - self.sphinx_size[0], window.screen.height - self.sphinx_size[1], *self.sphinx_size))
        pygame.draw.rect(widget_mask, (0, 0, 0, 255), (window.screen.width - 50, 0, 50, 50))

        self._bloom_mask = widget_mask
        self._laser_mask = laser_mask

    def draw(self):
        self._widget_group.draw()
        self.draw_sphinx()

        if self._fire_laser:
            window.set_apply_arguments(ShaderType.RAYS, occlusion=self._laser_mask, softShadow=0.1)

        window.set_apply_arguments(ShaderType.BLOOM, highlight_surface=self._bloom_mask, surface_intensity=0.3, brightness_intensity=0.6)

    def update(self, **kwargs):
        random_offset = lambda: randint(-5, 5) / 40
        if self._fire_laser:
            window.clear_effect(ShaderType.RAYS)
            window.set_effect(ShaderType.RAYS, lights=[[
                (self.sphinx_center[0] / window.size[0], self.sphinx_center[1] / window.size[1]),
                2.2,
                (190, 190, 255),
                0.99,
                (self.sphinx_rotation - 2 + random_offset(), self.sphinx_rotation + 2 + random_offset())
            ]])

            window.set_effect(ShaderType.SHAKE)
            window.set_apply_arguments(ShaderType.SHAKE, intensity=1)
            pygame.mouse.set_pos(pygame.mouse.get_pos()[0] + random_offset(), pygame.mouse.get_pos()[1] + random_offset())

        super().update(**kwargs)

    def handle_resize(self):
        super().handle_resize()
        self.update_masks()
```

```
from data.components.custom_event import CustomEvent
from data.constants import MenuEventType
```

```python
from data.managers.theme import theme
from data.assets import GRAPHICS
from data.widgets import *

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.15, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

MENU_WIDGETS = {
    'credits':
    Icon(
        relative_position=(0, 0),
        relative_size=(0.7, 0.7),
        icon=GRAPHICS['credits'],
        anchor_x='center',
        anchor_y='center',
        margin=50
    ),
    'default': [
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['quit_base'],
            hover_icon=GRAPHICS['quit_hover'],
            press_icon=GRAPHICS['quit_press'],
            event=CustomEvent(MenuEventType.QUIT_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['credits_base'],
            hover_icon=GRAPHICS['credits_hover'],
            press_icon=GRAPHICS['credits_press'],
            event=CustomEvent(MenuEventType.CREDITS_CLICK)
        ),
        ReactiveIconButton(
            relative_position=(0.05, -0.2),
            relative_size=(0, 0.15),
            anchor_y='center',
            base_icon=GRAPHICS['play_text_base'],
            hover_icon=GRAPHICS['play_text_hover'],
            press_icon=GRAPHICS['play_text_press'],
            event=CustomEvent(MenuEventType.CONFIG_CLICK)
        ),
        ReactiveIconButton(
            relative_position=(0.05, 0),
            relative_size=(0, 0.15),
            anchor_y='center',
            base_icon=GRAPHICS['review_text_base'],
            hover_icon=GRAPHICS['review_text_hover'],
            press_icon=GRAPHICS['review_text_press'],
            event=CustomEvent(MenuEventType.BROWSER_CLICK)
        ),
        ReactiveIconButton(
            relative_position=(0.05, 0.2),
            relative_size=(0, 0.15),
            anchor_y='center',
            base_icon=GRAPHICS['settings_text_base'],
            hover_icon=GRAPHICS['settings_text_hover'],
            press_icon=GRAPHICS['settings_text_press'],
            event=CustomEvent(MenuEventType.SETTINGS_CLICK)
        ),
        Icon(
            relative_position=(0.0, 0.1),
            relative_size=(0.3, 0.2),
            anchor_x='center',
            fill_colour=theme['fillSecondary'],
            icon=GRAPHICS['title_screen_art'],
            stretch=False
        ),
    ]
}

# Widgets used for testing light rays effect
TEST_WIDGETS = {
    'default': [
        Rectangle(
            relative_position=(0.4, 0.2),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.5, 0.7),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.6, 0.6),
            relative_size=(0.2, 0.2),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.4, 0.6),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.6, 0.4),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.3, 0.4),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.475, 0.15),
            relative_size=(0.2, 0.2),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        ),
        Rectangle(
            relative_position=(0.6, 0.2),
            relative_size=(0.1, 0.1),
            scale_mode='height',
            visible=True,
            border_width=0,
            fill_colour=(255, 0, 0),
            border_radius=1000
        )
    ]
}
```

```
import pygame
from collections import deque
from data.states.game.components.capture_draw import CaptureDraw
from data.states.game.components.piece_group import PieceGroup
from data.constants import ReviewEventType, Colour, ShaderType
from data.states.game.components.laser_draw import LaserDraw
from data.utils.bitboard_helpers import bitboard_to_coords
from data.states.review.widget_dict import REVIEW_WIDGETS
from data.utils.browser_helpers import get_winner_string
from data.states.game.components.board import Board
from data.components.game_entry import GameEntry
from data.managers.logs import initialise_logger
from data.managers.window import window
from data.control import _State
from data.assets import MUSIC

logger = initialise_logger(__name__)

class Review(_State):
    def __init__(self):
        super().__init__()

        self._moves = deque()
        self._popped_moves = deque()
        self._game_info = {}

        self._board = None
        self._piece_group = None
        self._laser_draw = None
        self._capture_draw = None

    def cleanup(self):
        """
        Cleanup function. Clears shader effects.
        """
        super().cleanup()

        window.clear_apply_arguments(ShaderType.BLOOM)
        window.clear_effect(ShaderType.RAYS)

        return None

    def startup(self, persist):
        """
        Startup function. Initialises all objects, widgets and game data.

        Args:
            persist (dict): Dict containing game entry data.
        """
        super().startup(REVIEW_WIDGETS, MUSIC['review'])

        window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.BACKGROUND_WAVES)
        window.set_apply_arguments(ShaderType.BLOOM, highlight_colours=[(pygame.Color('0x95e0cc')).rgb, pygame.Color('0xf14e52').rgb], colour_intensity=0.8)
        REVIEW_WIDGETS['help'].kill()

        self._moves = deque(GameEntry.parse_moves(persist.pop('moves', '')))
        self._popped_moves = deque()
        self._game_info = persist

        self._board = Board(self._game_info['start_fen_string'])
        self._piece_group = PieceGroup()
        self._laser_draw = LaserDraw(self.board_position, self.board_size)
        self._capture_draw = CaptureDraw(self.board_position, self.board_size)

        self.initialise_widgets()
        self.simulate_all_moves()
        self.refresh_pieces()
        self.refresh_widgets()

        self.draw()

    @property
    def board_position(self):
        return REVIEW_WIDGETS['chessboard'].position

    @property
    def board_size(self):
        return REVIEW_WIDGETS['chessboard'].size

    @property
    def square_size(self):
        return self.board_size[0] / 10

    def initialise_widgets(self):
        """
```

```python
        Initializes the widgets for a new game.
        """
        REVIEW_WIDGETS['move_list'].reset_move_list()
        REVIEW_WIDGETS['move_list'].kill()
        REVIEW_WIDGETS['scroll_area'].set_image()

        REVIEW_WIDGETS['winner_text'].set_text(f'WINNER: {get_winner_string(self._game_info["winner"])}')
        REVIEW_WIDGETS['blue_piece_display'].reset_piece_list()
        REVIEW_WIDGETS['red_piece_display'].reset_piece_list()

        if self._game_info['time_enabled']:
            REVIEW_WIDGETS['timer_disabled_text'].kill()
        else:
            REVIEW_WIDGETS['blue_timer'].kill()
            REVIEW_WIDGETS['red_timer'].kill()

    def refresh_widgets(self):
        """
        Refreshes the widgets after every move.
        """
        REVIEW_WIDGETS['move_number_text'].set_text(f'MOVE NO: {(len(self._moves)) / 2:.1f} / {(len(self._moves) + len(self._popped_moves)) / 2:.1f}')
        REVIEW_WIDGETS['move_colour_text'].set_text(f'{self.calculate_colour().name} TO MOVE')

        if self._game_info['time_enabled']:
            if len(self._moves) == 0:
                REVIEW_WIDGETS['blue_timer'].set_time(float(self._game_info['time']) * 60 * 1000)
                REVIEW_WIDGETS['red_timer'].set_time(float(self._game_info['time']) * 60 * 1000)
            else:
                REVIEW_WIDGETS['blue_timer'].set_time(float(self._moves[-1]['blue_time']) * 60 * 1000)
                REVIEW_WIDGETS['red_timer'].set_time(float(self._moves[-1]['red_time']) * 60 * 1000)

        REVIEW_WIDGETS['scroll_area'].set_image()

    def refresh_pieces(self):
        """
        Refreshes the pieces on the board.
        """
        self._piece_group.initialise_pieces(self._board.get_piece_list(), self.board_position, self.board_size)

    def simulate_all_moves(self):
        """
        Simulates all moves at the start of every game to obtain laser results and fill up piece display and move list widgets.
        """
        for index, move_dict in enumerate(self._moves):
            laser_result = self._board.apply_move(move_dict['move'], fire_laser=True)
            self._moves[index]['laser_result'] = laser_result

            if laser_result.hit_square_bitboard:
                if laser_result.piece_colour == Colour.BLUE:
                    REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit)
                elif laser_result.piece_colour == Colour.RED:
```

```python
                REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.piece_hit)

        REVIEW_WIDGETS['move_list'].append_to_move_list(move_dict['unparsed_move'])

def calculate_colour(self):
    """
    Calculates the current active colour to move.

    Returns:
        Colour: The current colour to move.
    """
    if self._game_info['start_fen_string'][-1].lower() == 'b':
        initial_colour = Colour.BLUE
    elif self._game_info['start_fen_string'][-1].lower() == 'r':
        initial_colour = Colour.RED

    if len(self._moves) % 2 == 0:
        return initial_colour
    else:
        return initial_colour.get_flipped_colour()

def handle_move(self, move, add_piece=True):
    """
    Handles applying or undoing a move.

    Args:
        move (dict): The move to handle.
        add_piece (bool): Whether to add the captured piece to the display. Defaults to True.
    """
    laser_result = move['laser_result']
    active_colour = self.calculate_colour()
    self._laser_draw.add_laser(laser_result, laser_colour=active_colour)

    if laser_result.hit_square_bitboard:
        if laser_result.piece_colour == Colour.BLUE:
            if add_piece:
                REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit)
            else:
                REVIEW_WIDGETS['red_piece_display'].remove_piece(laser_result.piece_hit)
        elif laser_result.piece_colour == Colour.RED:
            if add_piece:
                REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.piece_hit)
            else:
                REVIEW_WIDGETS['blue_piece_display'].remove_piece(laser_result.piece_hit)

        self._capture_draw.add_capture(
            laser_result.piece_hit,
            laser_result.piece_colour,
            laser_result.piece_rotation,
            bitboard_to_coords(laser_result.hit_square_bitboard),
            laser_result.laser_path[0][0],
            active_colour,
            shake=False
        )

    def update_laser_mask(self):
        """
        Updates the laser mask for the light rays effect.
        """
        temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
        self._piece_group.draw(temp_surface)
        mask = pygame.mask.from_surface(temp_surface, threshold=127)
        mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255, 0, 0, 255))

        window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)

    def get_event(self, event):
        """
        Processes Pygame events.

        Args:
            event (pygame.event.Event): The event to handle.
        """
        if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
            REVIEW_WIDGETS['help'].kill()

        widget_event = self._widget_group.process_event(event)

        if widget_event is None:
            return

        match widget_event.type:
            case None:
                return

            case ReviewEventType.MENU_CLICK:
                self.next = 'menu'
                self.done = True

            case ReviewEventType.PREVIOUS_CLICK:
                if len(self._moves) == 0:
                    return

                # Pop last applied move off first stack
                move = self._moves.pop()
                # Pushed onto second stack
                self._popped_moves.append(move)

                # Undo last applied move
                self._board.undo_move(move['move'], laser_result=move['laser_result'])
                self.handle_move(move, add_piece=False)
                REVIEW_WIDGETS['move_list'].pop_from_move_list()

                self.refresh_pieces()
                self.refresh_widgets()
                self.update_laser_mask()

            case ReviewEventType.NEXT_CLICK:
                if len(self._popped_moves) == 0:
                    return

                # Peek at second stack to get last undone move
                move = self._popped_moves[-1]

                # Reapply last undone move
                self._board.apply_move(move['move'])
                self.handle_move(move, add_piece=True)
                REVIEW_WIDGETS['move_list'].append_to_move_list(move['unparsed_move'])

                # Pop last undone move from second stack
                self._popped_moves.pop()
                # Push onto first stack
                self._moves.append(move)

                self.refresh_pieces()
                self.refresh_widgets()
                self.update_laser_mask()

            case ReviewEventType.HELP_CLICK:
                self._widget_group.add(REVIEW_WIDGETS['help'])
                self._widget_group.handle_resize(window.size)

    def handle_resize(self):
        """
        Handles resizing of the window.
        """
        super().handle_resize()
        self._piece_group.handle_resize(self.board_position, self.board_size)
        self._laser_draw.handle_resize(self.board_position, self.board_size)
        self._capture_draw.handle_resize(self.board_position, self.board_size)

        if self._laser_draw.firing:
            self.update_laser_mask()

    def draw(self):
        """
        Draws all components onto the window screen.
        """
        self._capture_draw.update()
        self._widget_group.draw()
        self._piece_group.draw(window.screen)
        self._laser_draw.draw(window.screen)
        self._capture_draw.draw(window.screen)
```

```
from data.widgets import *
from data.components.custom_event import CustomEvent
from data.constants import ReviewEventType, Colour
from data.assets import GRAPHICS

MOVE_LIST_WIDTH = 0.2

right_container = Rectangle(
    relative_position=(0.05, 0),
    relative_size=(0.2, 0.7),
    anchor_y='center',
    anchor_x='right'
)

info_container = Rectangle(
    parent=right_container,
    relative_position=(0, 0.5),
    relative_size=(1, 0.5),
    visible=True
)

arrow_container = Rectangle(
    relative_position=(0, 0.05),
    relative_size=(0.4, 0.1),
    anchor_x='center',
    anchor_y='bottom'
)

move_list = MoveList(
    parent=right_container,
    relative_position=(0, 0),
    relative_width=1,
    minimum_height=300,
    move_list=[]
)

top_right_container = Rectangle(
    relative_position=(0, 0),
    relative_size=(0.15, 0.075),
    fixed_position=(5, 5),
    anchor_x='right',
    scale_mode='height'
)

REVIEW_WIDGETS = {
    'help':
    Icon(
        relative_position=(0, 0),
        relative_size=(1.02, 1.02),
        icon=GRAPHICS['review_help'],
        anchor_x='center',
        anchor_y='center',
        border_width=0,
        fill_colour=(0, 0, 0, 0)
    ),
    'default': [
        arrow_container,
        right_container,
        info_container,
        top_right_container,
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            event=CustomEvent(ReviewEventType.MENU_CLICK)
        ),
        ReactiveIconButton(
            parent=top_right_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['help_base'],
            hover_icon=GRAPHICS['help_hover'],
            press_icon=GRAPHICS['help_press'],
            event=CustomEvent(ReviewEventType.HELP_CLICK)
        ),
        ReactiveIconButton(
            parent=arrow_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            base_icon=GRAPHICS['left_arrow_filled_base'],
            hover_icon=GRAPHICS['left_arrow_filled_hover'],
            press_icon=GRAPHICS['left_arrow_filled_press'],
            event=CustomEvent(ReviewEventType.PREVIOUS_CLICK)
        ),
        ReactiveIconButton(
            parent=arrow_container,
            relative_position=(0, 0),
            relative_size=(1, 1),
            scale_mode='height',
            anchor_x='right',
            base_icon=GRAPHICS['right_arrow_filled_base'],
            hover_icon=GRAPHICS['right_arrow_filled_hover'],
            press_icon=GRAPHICS['right_arrow_filled_press'],
            event=CustomEvent(ReviewEventType.NEXT_CLICK)
        ),
    ],
    'move_list':
        move_list,
    'scroll_area':
    ScrollArea(
        parent=right_container,
        relative_position=(0, 0),
        relative_size=(1, 0.5),
        vertical=True,
        widget=move_list
    ),
    'chessboard':
    Chessboard(
        relative_position=(0, 0),
        relative_width=0.4,
        scale_mode='width',
        anchor_x='center',
        anchor_y='center'
    ),
    'move_number_text':
    Text(
        parent=info_container,
        relative_position=(0, 0),
        relative_size=(1, 0.3),
        anchor_y='bottom',
        text='MOVE NO:',
        fit_vertical=False,
        margin=10,
        border_width=0,
        fill_colour=(0, 0, 0, 0),
    ),
    'move_colour_text':
    Text(
        parent=info_container,
        relative_size=(1, 0.3),
        relative_position=(0, 0),
        anchor_y='center',
        text='TO MOVE',
        fit_vertical=False,
        margin=10,
        border_width=0,
        fill_colour=(0, 0, 0, 0),
    ),
    'winner_text':
```

```
    Text(
        parent=info_container,
        relative_size=(1, 0.3),
        relative_position=(0, 0),
        text='WINNER:',
        fit_vertical=False,
        margin=10,
        border_width=0,
        fill_colour=(0, 0, 0, 0),
    ),
    'blue_timer':
    Timer(
        relative_position=(0.05, 0.05),
        anchor_y='center',
        relative_size=(0.1, 0.1),
        active_colour=Colour.BLUE,
    ),
    'red_timer':
    Timer(
        relative_position=(0.05, -0.05),
        anchor_y='center',
        relative_size=(0.1, 0.1),
        active_colour=Colour.RED
    ),
    'timer_disabled_text':
    Text(
        relative_size=(0.2, 0.1),
        relative_position=(0.05, 0),
        anchor_y='center',
        fit_vertical=False,
        text='TIMER DISABLED',
    ),
    'blue_piece_display':
    PieceDisplay(
        relative_position=(0.05, 0.05),
        relative_size=(0.2, 0.1),
        anchor_y='bottom',
        active_colour=Colour.BLUE
    ),
    'red_piece_display':
    PieceDisplay(
        relative_position=(0.05, 0.05),
        relative_size=(0.2, 0.1),
        active_colour=Colour.RED
    ),
}
```

```
import pygame
from random import randint
from data.utils.data_helpers import get_default_settings, get_user_settings,
    update_user_settings
from data.constants import SettingsEventType, WidgetState, ShaderType, SHADER_MAP
from data.states.settings.widget_dict import SETTINGS_WIDGETS
from data.managers.logs import initialise_logger
from data.managers.window import window
from data.managers.audio import audio
from data.widgets import ColourPicker
from data.control import _State
from data.assets import MUSIC

logger = initialise_logger(__name__)

```

```python
class Settings(_State):
    def __init__(self):
        super().__init__()

        self._colour_picker = None
        self._settings = None

    def cleanup(self):
        super().cleanup()

        update_user_settings(self._settings)

        return None

    def startup(self, persist=None):
        super().startup(SETTINGS_WIDGETS, music=MUSIC[f'menu_{randint(1, 3)}'])

        window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.BACKGROUND_BALATRO)
        self._settings = get_user_settings()
        self.reload_settings()

        self.draw()

    def create_colour_picker(self, mouse_pos, button_type):
        if button_type == SettingsEventType.PRIMARY_COLOUR_BUTTON_CLICK:
            selected_colour = self._settings['primaryBoardColour']
            event_type = SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK
        else:
            selected_colour = self._settings['secondaryBoardColour']
            event_type = SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK

        self._colour_picker = ColourPicker(
            relative_position=(mouse_pos[0] / window.size[0], mouse_pos[1] / window.size[1]),
            relative_width=0.15,
            selected_colour=selected_colour,
            event_type=event_type
        )
        self._widget_group.add(self._colour_picker)

    def remove_colour_picker(self):
        self._colour_picker.kill()

    def reload_display_mode(self):
        relative_mouse_pos = (pygame.mouse.get_pos()[0] / window.size[0], pygame.mouse.get_pos()[1] / window.size[1])

        if self._settings['displayMode'] == 'fullscreen':
            window.set_fullscreen(desktop=True)
            window.handle_resize()

        elif self._settings['displayMode'] == 'windowed':
            window.set_windowed()
            window.handle_resize()
            window.restore()

        self._widget_group.handle_resize(window.size)

        new_mouse_pos = (relative_mouse_pos[0] * window.size[0], relative_mouse_pos[1] * window.size[1])
        pygame.mouse.set_pos(new_mouse_pos)

    def reload_shaders(self):
        window.clear_all_effects()

        for shader_type in SHADER_MAP[self._settings['shader']]:
            window.set_effect(shader_type)

    def reload_settings(self):
        SETTINGS_WIDGETS['primary_colour_button'].initialise_new_colours(self._settings['primaryBoardColour'])
        SETTINGS_WIDGETS['secondary_colour_button'].initialise_new_colours(self._settings['secondaryBoardColour'])
        SETTINGS_WIDGETS['primary_colour_button'].set_state_colour(WidgetState.BASE)
        SETTINGS_WIDGETS['secondary_colour_button'].set_state_colour(WidgetState.BASE)
        SETTINGS_WIDGETS['music_volume_slider'].set_volume(self._settings['musicVolume'])
        SETTINGS_WIDGETS['sfx_volume_slider'].set_volume(self._settings['sfxVolume'])
        SETTINGS_WIDGETS['display_mode_dropdown'].set_selected_word(self._settings['displayMode'])
        SETTINGS_WIDGETS['shader_carousel'].set_to_key(self._settings['shader'])
        SETTINGS_WIDGETS['particles_switch'].set_toggle_state(self._settings['particles'])
        SETTINGS_WIDGETS['opengl_switch'].set_toggle_state(self._settings['opengl'])

        self.reload_shaders()
        self.reload_display_mode()

    def get_event(self, event):
        widget_event = self._widget_group.process_event(event)

        if widget_event is None:
            if event.type == pygame.MOUSEBUTTONDOWN and self._colour_picker:
                self.remove_colour_picker()
            return

        match widget_event.type:
            case SettingsEventType.VOLUME_SLIDER_SLIDE:
                return

            case SettingsEventType.VOLUME_SLIDER_CLICK:
                if widget_event.volume_type == 'music':
                    audio.set_music_volume(widget_event.volume)
                    self._settings['musicVolume'] = widget_event.volume
                elif widget_event.volume_type == 'sfx':
                    audio.set_sfx_volume(widget_event.volume)
                    self._settings['sfxVolume'] = widget_event.volume

            case SettingsEventType.DROPDOWN_CLICK:
                selected_word = SETTINGS_WIDGETS['display_mode_dropdown'].get_selected_word()

                if selected_word is None or selected_word == self._settings['displayMode']:
                    return

                self._settings['displayMode'] = selected_word

                self.reload_display_mode()

            case SettingsEventType.MENU_CLICK:
                self.next = 'menu'
                self.done = True

            case SettingsEventType.RESET_DEFAULT:
                self._settings = get_default_settings()
                self.reload_settings()

            case SettingsEventType.RESET_USER:
                self._settings = get_user_settings()
                self.reload_settings()

            case SettingsEventType.PRIMARY_COLOUR_BUTTON_CLICK | SettingsEventType.SECONDARY_COLOUR_BUTTON_CLICK:
                if self._colour_picker:
                    self.remove_colour_picker()

                self.create_colour_picker(event.pos, widget_event.type)

            case SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK | SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK:
                if widget_event.colour:
                    r, g, b = widget_event.colour.rgb
                    hex_colour = f'0x{hex(r)[2:].zfill(2)}{hex(g)[2:].zfill(2)}{hex(b)[2:].zfill(2)}'

                    if widget_event.type == SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK:
                        SETTINGS_WIDGETS['primary_colour_button'].initialise_new_colours(widget_event.colour)
                        SETTINGS_WIDGETS['primary_colour_button'].set_state_colour(WidgetState.BASE)
                        self._settings['primaryBoardColour'] = hex_colour
                    elif widget_event.type == SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK:
                        SETTINGS_WIDGETS['secondary_colour_button'].initialise_new_colours(widget_event.colour)
                        SETTINGS_WIDGETS['secondary_colour_button'].set_state_colour(WidgetState.BASE)
                        self._settings['secondaryBoardColour'] = hex_colour

            case SettingsEventType.SHADER_PICKER_CLICK:
                self._settings['shader'] = widget_event.data
                self.reload_shaders()

            case SettingsEventType.OPENGL_CLICK:
                self._settings['opengl'] = widget_event.toggled
                self.reload_shaders()

            case SettingsEventType.PARTICLES_CLICK:
                self._settings['particles'] = widget_event.toggled

    def draw(self):
        self._widget_group.draw()
```

```python
from data.widgets import *
from data.components.custom_event import CustomEvent
from data.constants import SettingsEventType, SHADER_MAP
from data.utils.data_helpers import get_user_settings
from data.assets import GRAPHICS, DEFAULT_FONT
from data.managers.theme import theme
from data.utils.font_helpers import text_width_to_font_size
from data.managers.window import window

user_settings = get_user_settings()
# font_size = text_width_to_font_size('Shaders (OPENGL GPU REQUIRED)',
    DEFAULT_FONT, 0.4 * window.screen.width)
FONT_SIZE = 21

carousel_widgets = {
    key: Text(
        relative_position=(0, 0),
        relative_size=(0.25, 0.04),
        margin=0,
        text=key.replace('_', ' ').upper(),
        fit_vertical=True,
        border_width=0,
        fill_colour=(0, 0, 0, 0),
    ) for key in SHADER_MAP.keys()
}

reset_container = Rectangle(
    relative_size=(0.2, 0.2),
    relative_position=(0, 0),
    fixed_position=(5, 5),
    anchor_x='right',
    anchor_y='bottom',
)

SETTINGS_WIDGETS = {
    'default': [
        reset_container,
        ReactiveIconButton(
            relative_position=(0, 0),
            relative_size=(0.075, 0.075),
            anchor_x='right',
            scale_mode='height',
            base_icon=GRAPHICS['home_base'],
            hover_icon=GRAPHICS['home_hover'],
            press_icon=GRAPHICS['home_press'],
            fixed_position=(5, 5),
            event=CustomEvent(SettingsEventType.MENU_CLICK)
        ),
        Text(
            relative_position=(0.01, 0.1),
            text='Display mode',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.2),
            text='Music',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.3),
            text='SFX',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.4),
            text='Primary board colour',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.5),
            text='Secondary board colour',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.6),
            text='Particles',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.7),
            text='Shaders (OPENGL GPU REQUIRED)',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        Text(
            relative_position=(0.01, 0.8),
            text='Super Secret Settings',
            relative_size=(0.4, 0.04),
            center=False,
            border_width=0,
            margin=0,
            font_size=21,
            fill_colour=(0, 0, 0, 0)
        ),
        TextButton(
            parent=reset_container,
            relative_position=(0, 0),
            relative_size=(1, 0.5),
            fit_vertical=False,
            margin=10,
            text='DISCARD CHANGES',
            text_colour=theme['textSecondary'],
            event=CustomEvent(SettingsEventType.RESET_USER)
        ),
        TextButton(
            parent=reset_container,
            relative_position=(0, 0.5),
            relative_size=(1, 0.5),
            fit_vertical=False,
            margin=10,
            text='RESET TO DEFAULT',
            text_colour=theme['textSecondary'],
            event=CustomEvent(SettingsEventType.RESET_DEFAULT)
        )
    ],
    'display_mode_dropdown':
    Dropdown(
        relative_position=(0.4, 0.1),
        relative_width=0.2,
        word_list=['fullscreen', 'windowed'],
        fill_colour=(255, 100, 100),
        event=CustomEvent(SettingsEventType.DROPDOWN_CLICK)
    ),
    'primary_colour_button':
    ColourButton(
        relative_position=(0.4, 0.4),
        relative_size=(0.08, 0.05),
        fill_colour=user_settings['primaryBoardColour'],
        border_width=5,
        event=CustomEvent(SettingsEventType.PRIMARY_COLOUR_BUTTON_CLICK)
    ),
    'secondary_colour_button':
    ColourButton(
        relative_position=(0.4, 0.5),
        relative_size=(0.08, 0.05),
        fill_colour=user_settings['secondaryBoardColour'],
        border_width=5,
        event=CustomEvent(SettingsEventType.SECONDARY_COLOUR_BUTTON_CLICK)
    ),
    'music_volume_slider':
    VolumeSlider(
        relative_position=(0.4, 0.2),
        relative_length=(0.5),
        default_volume=user_settings['musicVolume'],
        border_width=5,
        volume_type='music'
    ),
    'sfx_volume_slider':
    VolumeSlider(
        relative_position=(0.4, 0.3),
        relative_length=(0.5),
        default_volume=user_settings['sfxVolume'],
        border_width=5,
        volume_type='sfx'
    ),
    'shader_carousel':
    Carousel(
        relative_position = (0.4, 0.8),
        margin=5,
        border_width=0,
        fill_colour=(0, 0, 0, 0),
        widgets_dict=carousel_widgets,
        event=CustomEvent(SettingsEventType.SHADER_PICKER_CLICK),
    ),
    'particles_switch':
    Switch(
        relative_position=(0.4, 0.6),
        relative_height=0.04,
        event=CustomEvent(SettingsEventType.PARTICLES_CLICK)
    ),
    'opengl_switch':
    Switch(
        relative_position=(0.4, 0.7),
        relative_height=0.04,
        event=CustomEvent(SettingsEventType.OPENGL_CLICK)
    ),
}
```

```
import pygame
from PIL import Image
from functools import cache
from random import sample, randint
import math

@cache
def scale_and_cache(image, target_size):
    """
    Caches image when resized repeatedly.

    Args:
        image (pygame.Surface): Image surface to be resized.
        target_size (tuple[float, float]): New image size.

    Returns:
        pygame.Surface: Resized image surface.
    """
    return pygame.transform.scale(image, target_size)

@cache
def smoothscale_and_cache(image, target_size):
    """
    Same as scale_and_cache, but with the Pygame smoothscale function.

    Args:
        image (pygame.Surface): Image surface to be resized.
        target_size (tuple[float, float]): New image size.

    Returns:
        pygame.Surface: Resized image surface.
    """
    return pygame.transform.smoothscale(image, target_size)

def gif_to_frames(path):
    """
    Uses the PIL library to break down GIFs into individual frames.

    Args:
        path (str): Directory path to GIF file.

    Yields:
```

```python
        PIL.Image: Single frame.
    """
    try:
        image = Image.open(path)

        first_frame = image.copy().convert('RGBA')
        yield first_frame
        image.seek(1)

        while True:
            current_frame = image.copy()
            yield current_frame
            image.seek(image.tell() + 1)
    except EOFError:
        pass

def get_perimeter_sample(image_size, number):
    """
    Used for particle drawing class, generates roughly equally distributed points around a rectangular image surface's perimeter.

    Args:
        image_size (tuple[float, float]): Image surface size.
        number (int): Number of points to be generated.

    Returns:
        list[tuple[int, int], ...]: List of random points on perimeter of image surface.
    """
    perimeter = 2 * (image_size[0] + image_size[1])
    # Flatten perimeter to a single number representing the distance from the top-middle of the surface going clockwise, and create a list of equally spaced points
    perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in range(0, number)]
    pos_list = []

    for perimeter_offset in perimeter_offsets:
        # For every point, add a random offset
        max_displacement = int(perimeter / (number * 4))
        perimeter_offset += randint(-max_displacement, max_displacement)

        if perimeter_offset > perimeter:
            perimeter_offset -= perimeter

        # Convert 1D distance back into 2D points on image surface perimeter
        if perimeter_offset < image_size[0]:
            pos_list.append((perimeter_offset, 0))
        elif perimeter_offset < image_size[0] + image_size[1]:
            pos_list.append((image_size[0], perimeter_offset - image_size[0]))
        elif perimeter_offset < image_size[0] + image_size[1] + image_size[0]:
            pos_list.append((perimeter_offset - image_size[0] - image_size[1], image_size[1]))
        else:
            pos_list.append((0, perimeter - perimeter_offset))
    return pos_list

def get_angle_between_vectors(u, v, deg=True):
    """
    Uses the dot product formula to find the angle between two vectors.

    Args:
        u (list[int, int]): Vector 1.
        v (list[int, int]): Vector 2.
        deg (bool, optional): Return results in degrees. Defaults to True.

    Returns:
        float: Angle between vectors.
    """
    dot_product = sum(i * j for (i, j) in zip(u, v))
    u_magnitude = math.sqrt(u[0] ** 2 + u[1] ** 2)
    v_magnitude = math.sqrt(v[0] ** 2 + v[1] ** 2)

    cos_angle = dot_product / (u_magnitude * v_magnitude)
    radians = math.acos(min(max(cos_angle, -1), 1))

    if deg:
        return math.degrees(radians)
    else:
        return radians

def get_rotational_angle(u, v, deg=True):
    """
    Get bearing angle relative to positive x-axis centered on second vector.

    Args:
        u (list[int, int]): Vector 1.
        v (list[int, int]): Vector 2, set as center of axes.
        deg (bool, optional): Return results in degrees. Defaults to True.

    Returns:
        float: Bearing angle between vectors.
    """
    radians = math.atan2(u[1] - v[1], u[0] -v[0])

    if deg:
        return math.degrees(radians)
    else:
        return radians

def get_vector(src_vertex, dest_vertex):
    """
    Get vector describing translation between two points.

    Args:
        src_vertex (list[int, int]): Source vertex.
        dest_vertex (list[int, int]): Destination vertex.

    Returns:
        tuple[int, int]: Vector between the two points.
    """
    return (dest_vertex[0] - src_vertex[0], dest_vertex[1] - src_vertex[1])

def get_next_corner(vertex, image_size):
    """
    Used in particle drawing system, finds coordinates of the next corner going clockwise, given a point on the perimeter.

    Args:
        vertex (list[int, int]): Point on perimeter.
        image_size (list[int, int]): Image size.

    Returns:
        list[int, int]: Coordinates of corner on perimeter.
```

```python
    """
    corners = [(0, 0), (image_size[0], 0), (image_size[0], image_size[1]), (0, image_size[1])]

    if vertex in corners:
        return corners[(corners.index(vertex) + 1) % len(corners)]

    if vertex[1] == 0:
        return (image_size[0], 0)
    elif vertex[0] == image_size[0]:
        return image_size
    elif vertex[1] == image_size[1]:
        return (0, image_size[1])
    elif vertex[0] == 0:
        return (0, 0)

def pil_image_to_surface(pil_image):
    """
    Args:
        pil_image (PIL.Image): Image to be converted.

    Returns:
        pygame.Surface: Converted image surface.
    """
    return pygame.image.frombytes(pil_image.tobytes(), pil_image.size, pil_image.mode).convert()

def calculate_frame_index(elapsed_milliseconds, start_index, end_index, fps):
    """
    Determine frame of animated GIF to be displayed.

    Args:
        elapsed_milliseconds (int): Milliseconds since GIF started playing.
        start_index (int): Start frame of GIF.
        end_index (int): End frame of GIF.
        fps (int): Number of frames to be played per second.

    Returns:
        int: Displayed frame index of GIF.
    """
    ms_per_frame = int(1000 / fps)
    return start_index + ((elapsed_milliseconds // ms_per_frame) % (end_index - start_index))

def draw_background(screen, background, current_time=0):
    """
    Draws background to screen

    Args:
        screen (pygame.Surface): Screen to be drawn to
        background (list[pygame.Surface, ...] | pygame.Surface): Background to be drawn, if GIF, list of surfaces indexed to select frame to be drawn
        current_time (int, optional): Used to calculate frame index for GIF. Defaults to 0.
    """
    if isinstance(background, list):
        # Animated background passed in as list of surfaces, calculate_frame_index() used to get index of frame to be drawn
        frame_index = calculate_frame_index(current_time, 0, len(background), fps=8)
        scaled_background = scale_and_cache(background[frame_index], screen.size)
        screen.blit(scaled_background, (0, 0))
    else:
        scaled_background = scale_and_cache(background, screen.size)
        screen.blit(scaled_background, (0, 0))

def get_highlighted_icon(icon):
    """
    Used for pressable icons, draws overlay on icon to show as pressed.

    Args:
        icon (pygame.Surface): Icon surface.

    Returns:
        pygame.Surface: Icon with overlay drawn on top.
    """
    icon_copy = icon.copy()
    overlay = pygame.Surface((icon.get_width(), icon.get_height()), pygame.SRCALPHA)
    overlay.fill((0, 0, 0, 128))
    icon_copy.blit(overlay, (0, 0))
    return icon_copy
```

```python
from data.constants import Rank, File, EMPTY_BB
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

def print_bitboard(bitboard):
    if (bitboard >= (2 ** 80)):
        raise ValueError('Invalid bitboard: too many bits')

    characters = ''
    for rank in reversed(Rank):

        for file in File:
            mask = 1 << (rank * 10 + file)
            if (bitboard & mask) != 0:
                characters += '1  '
            else:
                characters += '.  '

        characters += '\n\n'

    logger.info('\n' + characters + '\n')

def is_occupied(bitboard, target_bitboard):
    return (target_bitboard & bitboard) != EMPTY_BB

def clear_square(bitboard, target_bitboard):
    return (~target_bitboard & bitboard)

def set_square(bitboard, target_bitboard):
    return (target_bitboard | bitboard)

def index_to_bitboard(index):
    return (1 << index)

def coords_to_bitboard(coords):
    index = coords[1] * 10 + coords[0]
    return index_to_bitboard(index)

def bitboard_to_notation(bitboard):
    index = bitboard_to_index(bitboard)
    x = index // 10
    y = index % 10

    return chr(y + 97) + str(x + 1)

def notation_to_bitboard(notation):
    index = (int(notation[1]) - 1) * 10 + int(ord(notation[0])) - 97

    return index_to_bitboard(index)

def bitboard_to_index(bitboard):
    return bitboard.bit_length() - 1

def bitboard_to_coords(bitboard):
    list_position = bitboard_to_index(bitboard)
    x = list_position % 10
    y = list_position // 10

    return x, y

def bitboard_to_coords_list(bitboard):
    list_positions = []

    for square in occupied_squares(bitboard):
        list_positions.append(bitboard_to_coords(square))

    return list_positions

def occupied_squares(bitboard):
    while bitboard:
        lsb_square = bitboard & -bitboard
        bitboard = bitboard ^ lsb_square

        yield lsb_square

def pop_count(bitboard):
    count = 0
    while bitboard:
        count += 1
        lsb_square = bitboard & -bitboard
        bitboard = bitboard ^ lsb_square

    return count

# def pop_count(bitboard):
#     count = 0
#     while bitboard:
#         count += 1
#         bitboard &= bitboard - 1

#     return count

def loop_all_squares():
    for i in range(80):
        yield 1 << i

#Solar
def get_LSB_value(bitboard: int):
    return bitboard & -bitboard

def pop_count_2(bitboard):
    count = 0
    while bitboard > 0:
        lsb_value = get_LSB_value(bitboard)
        count += 1
        bitboard ^= lsb_value

    return count
```

```
import pygame
from data.utils.data_helpers import get_user_settings
from data.assets import DEFAULT_FONT

user_settings = get_user_settings()

def create_board(board_size, primary_colour, secondary_colour, font=DEFAULT_FONT):
    square_size = board_size[0] / 10
    board_surface = pygame.Surface(board_size)

    for i in range(80):
        x = i % 10
        y = i // 10

        if (x + y) % 2 == 0:
            square_colour = primary_colour
        else:
            square_colour = secondary_colour

        square_x = x * square_size
        square_y = y * square_size

        pygame.draw.rect(board_surface, square_colour, (square_x, square_y,
    square_size + 1, square_size + 1)) # +1 to fill in black lines

        if y == 7:
            text_position = (square_x + square_size * 0.7, square_y + square_size
    * 0.55)
            text_size = square_size / 3
            font.render_to(board_surface, text_position, str(chr(x + 1 + 96)),
    fgcolor=(10, 10, 10, 175), size=text_size)
        if x == 0:
            text_position = (square_x + square_size * 0.1, square_y + square_size
    * 0.1)
            text_size = square_size / 3
            font.render_to(board_surface, text_position, str(7-y + 1), fgcolor
    =(10, 10, 10, 175), size=text_size)

    return board_surface

def create_square_overlay(square_size, colour):
    overlay = pygame.Surface((square_size, square_size), pygame.SRCALPHA)
    overlay.fill(colour)

    return overlay

def create_circle_overlay(square_size, colour):
    overlay = pygame.Surface((square_size, square_size), pygame.SRCALPHA)
    pygame.draw.circle(overlay, colour, (square_size / 2, square_size / 2),
    square_size / 4)

    return overlay

def coords_to_screen_pos(coords, board_position, square_size):
    x = board_position[0] + (coords[0] * square_size)
```

```python
    y = board_position[1] + ((7 - coords[1]) * square_size)

    return (x, y)

def screen_pos_to_coords(mouse_position, board_position, board_size):
    if (board_position[0] <= mouse_position[0] <= board_position[0] + board_size[0]) and (board_position[1] <= mouse_position[1] <= board_position[1] + board_size[1]):
        x = (mouse_position[0] - board_position[0]) // (board_size[0] / 10)
        y = (board_size[1] - (mouse_position[1] - board_position[1])) // (board_size[0] / 10)
        return (int(x), int(y))

    return None
```

```python
from data.constants import Miscellaneous, Colour

def get_winner_string(winner):
    if winner is None:
        return 'UNFINISHED'
    elif winner == Miscellaneous.DRAW:
        return 'DRAW'
    else:
        return Colour(winner).name
```

```python
import sqlite3
from pathlib import Path
from datetime import datetime

database_path = (Path(__file__).parent / '../database/database.db').resolve()

def insert_into_games(game_entry):
    """
    Inserts a new row into games table.

    Args:
        game_entry (GameEntry): GameEntry object containing game information.
    """
    connection = sqlite3.connect(database_path, detect_types=sqlite3.PARSE_DECLTYPES)
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()

    # Datetime added for created_dt column
    game_entry = (*game_entry, datetime.now())

    cursor.execute('''
        INSERT INTO games (cpu_enabled, cpu_depth, winner, time_enabled, time, number_of_ply, moves, start_fen_string, final_fen_string, created_dt)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    ''', game_entry)

    connection.commit()

    # Return inserted row
    cursor.execute('''
        SELECT * FROM games WHERE id = LAST_INSERT_ROWID()
    ''')
    inserted_row = cursor.fetchone()

    connection.close()

    return dict(inserted_row)

def get_all_games():
    """
    Get all rows in games table.

    Returns:
        list[dict]: List of game entries represented as dictionaries.
    """
    connection = sqlite3.connect(database_path, detect_types=sqlite3.PARSE_DECLTYPES)
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()

    cursor.execute('''
        SELECT * FROM games
    ''')
    games = cursor.fetchall()

    connection.close()

    return [dict(game) for game in games]

def delete_all_games():
    """
    Delete all rows in games table.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        DELETE FROM games
    ''')

    connection.commit()
    connection.close()

def delete_game(id):
    """
    Deletes specific row in games table using id attribute.

    Args:
        id (int): Primary key for row.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute('''
        DELETE FROM games WHERE id = ?
    ''', (id,))

    connection.commit()
    connection.close()

def get_ordered_games(column, ascend=True, start_row=1, end_row=10):
    """
    Get specific number of rows from games table ordered by a specific column(s).

    Args:
        column (_type_): Column to sort by.
        ascend (bool, optional): Sort ascending or descending. Defaults to True.
        start_row (int, optional): First row returned. Defaults to 1.
        end_row (int, optional): Last row returned. Defaults to 10.

    Raises:
        ValueError: If ascend argument or column argument are invalid types.

    Returns:
        list[dict]: List of ordered game entries represented as dictionaries.
    """
    if not isinstance(ascend, bool) or not isinstance(column, str):
        raise ValueError('(database_helpers.get_ordered_games) Invalid input arguments!')

    connection = sqlite3.connect(database_path, detect_types=sqlite3.PARSE_DECLTYPES)
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()

    # Match ascend bool to correct SQL keyword
    if ascend:
        ascend_arg = 'ASC'
    else:
        ascend_arg = 'DESC'

    # Partition by winner, then order by time and number_of_ply
    if column == 'winner':
        cursor.execute(f'''
            SELECT * FROM
                (SELECT ROW_NUMBER() OVER (
                    PARTITION BY winner
                    ORDER BY time {ascend_arg}, number_of_ply {ascend_arg}
                ) AS row_num, * FROM games)
            WHERE row_num >= ? AND row_num <= ?
        ''', (start_row, end_row))
    else:
    # Order by time or number_of_ply only
        cursor.execute(f'''
            SELECT * FROM
                (SELECT ROW_NUMBER() OVER (
                    ORDER BY {column} {ascend_arg}
                ) AS row_num, * FROM games)
            WHERE row_num >= ? AND row_num <= ?
        ''', (start_row, end_row))

    games = cursor.fetchall()

    connection.close()

    return [dict(game) for game in games]

def get_number_of_games():
    """
    Returns:
        int: Number of rows in the games.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

    cursor.execute("""
        SELECT COUNT(ROWID) FROM games
    """)

    result = cursor.fetchall()[0][0]

    connection.close()

    return result

# delete_all_games()
```

```
import json
from pathlib import Path

module_path = Path(__file__).parent
default_file_path = (module_path / '../app_data/default_settings.json').resolve()
user_file_path = (module_path / '../app_data/user_settings.json').resolve()
themes_file_path = (module_path / '../app_data/themes.json').resolve()

def load_json(path):
    """
    Args:
        path (str): Path to JSON file.

    Raises:
        Exception: Invalid file.

    Returns:
        dict: Parsed JSON file.
    """
    try:
        with open(path, 'r') as f:
            file = json.load(f)

        return file
    except:
        raise Exception('Invalid JSON file (data_helpers.py)')

def get_user_settings():
    return load_json(user_file_path)

def get_default_settings():
    return load_json(default_file_path)

def get_themes():
    return load_json(themes_file_path)

def update_user_settings(data):
    """
    Rewrites JSON file for user settings with new data.

    Args:
        data (dict): Dictionary storing updated user settings.

    Raises:
        Exception: Invalid file.
    """
    try:
        with open(user_file_path, 'w') as f:
            json.dump(data, f, indent=4)
    except:
        raise Exception('Invalid JSON file (data_helpers.py)')
```

```
def height_to_font_size(font, target_height):
    test_size = 1
    while True:
        glyph_metrics = font.get_metrics('j', size=test_size)
        descender = font.get_sized_descender(test_size)
        test_height = abs(glyph_metrics[0][3] - glyph_metrics[0][2]) - descender
        if test_height > target_height:
            return test_size - 1

        test_size += 1

def width_to_font_size(font, target_width):
    test_size = 1
    while True:
        glyph_metrics = font.get_metrics(' ', size=test_size)

        if (glyph_metrics[0][4] * 8) > target_width:
            return (test_size - 1)

        test_size += 1

def text_width_to_font_size(text, font, target_width):
    test_size = 1
    if len(text) == 0:
        # print('(text_width_to_font_size) Text must have length greater than 1!')
        text = " "

    while True:
        text_rect = font.get_rect(text, size=test_size)

        if text_rect.width > target_width:
            return (test_size - 1)

        test_size += 1

def text_height_to_font_size(text, font, target_height):
    test_size = 1

    if ('(' in text) or (')' in text):
        text = text.replace('(', 'j') # Pygame freetype thinks '(' or ')' is taller for some reason
        text = text.replace(')', 'j')

    if len(text) == 0:
        # print('(text_height_to_font_size) Text must have length greater than 1!')
        text = "j"

    while True:
        text_rect = font.get_rect(text, size=test_size)

        if text_rect.height > target_height:
            return (test_size - 1)

        test_size += 1

def get_font_height(font, font_size):
    glyph_metrics = font.get_metrics('j', size=font_size)
    descender = font.get_sized_descender(font_size)
    return abs(glyph_metrics[0][3] - glyph_metrics[0][2]) - descender
```

```
from data.constants import MoveType, Rotation

def parse_move_type(move_type):
    if move_type.isalpha() is False:
```

```python
        raise ValueError('Invalid move type - move type must be a string!')
    if move_type.lower() not in MoveType:
        raise ValueError('Invalid move - type - move type must be m or r!')

    return MoveType(move_type.lower())

def parse_notation(notation):
    if (notation[0].isalpha() is False) or (notation[1].isnumeric() is False):
        raise ValueError('Invalid notation - invalid notation input types!')
    if not (97 <= ord(notation[0]) <= 106):
        raise ValueError('Invalid notation - file is out of range!')
    elif not (0 <= int(notation[1]) <= 10):
        raise ValueError('Invalid notation - rank is out of range!')

    return notation

def parse_rotation(rotation):
    if rotation == '':
        return None
    if rotation.isalpha() is False:
        raise ValueError('Invalid rotation - rotation must be a string!')
    if rotation.lower() not in Rotation:
        raise ValueError('Invalid rotation - rotation is invalid!')

    return Rotation(rotation.lower())
```

```python
import pygame
from pathlib import Path

import pygame.freetype
from data.utils.asset_helpers import gif_to_frames, pil_image_to_surface

def convert_gfx_alpha(image, colorkey=(0, 0, 0)):
    # if image.get_alpha():
        return image.convert_alpha()
    # else:
    #    image = image.convert_alpha()
    #    image.set_colorkey(colorkey)

    #    return image

def load_gfx(path, colorkey=(0, 0, 0), accept=(".svg", ".png", ".jpg", ".gif")):
    file_path = Path(path)
    name, extension = file_path.stem, file_path.suffix

    if extension.lower() in accept:
        if extension.lower() == '.gif':
            frames_list = []

            for frame in gif_to_frames(path):
                image_surface = pil_image_to_surface(frame)
                frames_list.append(image_surface)

            return frames_list

        if extension.lower() == '.svg':
            low_quality_image = pygame.image.load_sized_svg(path, (200, 200))
            image = pygame.image.load(path)
            image = convert_gfx_alpha(image, colorkey)

            return [image, low_quality_image]

        else:
            image = pygame.image.load(path)
            return convert_gfx_alpha(image, colorkey)

def load_all_gfx(directory, colorkey=(0, 0, 0), accept=(".svg", ".png", ".jpg", ".gif")):
    graphics = {}

    for file in Path(directory).rglob('*'):
        name, extension = file.stem, file.suffix
        path = Path(directory / file)

        if extension.lower() in accept and 'old' not in name:
            if name == 'piece_spritesheet':
                data = load_spritesheet(
                    path,
                    (16, 16),
                    ['pyramid_1', 'scarab_1', 'anubis_1', 'pharoah_1', 'sphinx_1', 'pyramid_0', 'scarab_0', 'anubis_0', 'pharoah_0', 'sphinx_0'],
                    ['_a', '_b', '_c', '_d'])

                graphics = graphics | data
                continue

            data = load_gfx(path, colorkey, accept)

            if isinstance(data, list):
                graphics[name] = data[0]
                graphics[f'{name}_lq'] = data[1]
            else:
                graphics[name] = data

    return graphics

def load_spritesheet(path, sprite_size, col_names, row_names):
    spritesheet = load_gfx(path)
    col_count = int(spritesheet.width / sprite_size[0])
    row_count = int(spritesheet.height / sprite_size[1])

    sprite_dict = {}

    for column in range(col_count):
        for row in range(row_count):
            surface = pygame.Surface(sprite_size, pygame.SRCALPHA)
            name = col_names[column] + row_names[row]

            surface.blit(spritesheet, (0, 0), (column * sprite_size[0], row * sprite_size[1], *sprite_size))
            sprite_dict[name] = surface

    return sprite_dict

def load_all_fonts(directory, accept=(".ttf", ".otf")):
    fonts = {}

    for file in Path(directory).rglob('*'):
        name, extension = file.stem, file.suffix
        path = Path(directory / file)

        if extension.lower() in accept:
            font = pygame.freetype.Font(path)
            fonts[name] = font
```

```python

    return fonts

def load_all_sfx(directory, accept=(".mp3", ".wav", ".ogg")):
    sound_effects = {}

    for file in Path(directory).rglob('*'):
        name, extension = file.stem, file.suffix
        path = Path(directory / file)

        if extension.lower() in accept and 'old' not in name:
            sound_effects[name] = load_sfx(path)

    return sound_effects

def load_sfx(path, accept=(".mp3", ".wav", ".ogg")):
    file_path = Path(path)
    name, extension = file_path.stem, file_path.suffix

    if extension.lower() in accept:
        sfx = pygame.mixer.Sound(path)
        return sfx

def load_all_music(directory, accept=(".mp3", ".wav", ".ogg")):
    music_paths = {}
    for file in Path(directory).rglob('*'):
        name, extension = file.stem, file.suffix
        path = Path(directory / file)

        if extension.lower() in accept:
            music_paths[name] = path

    return music_paths
```

```python
import pygame
from math import sqrt

def create_slider(size, fill_colour, border_width, border_colour):
    """
    Creates surface for sliders.

    Args:
        size (list[int, int]): Image size.
        fill_colour (pygame.Color): Fill (inner) colour.
        border_width (float): Border width.
        border_colour (pygame.Color): Border colour.

    Returns:
        pygame.Surface: Slider image surface.
    """
    gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
    border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.height))

    # Draws rectangle with a border radius half of image height, to draw an rectangle with semicurclar cap (obround)
    pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int(size[1] / 2))
    pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(border_width), border_radius=int(size[1] / 2))

    return gradient_surface
```

```python

def create_slider_gradient(size, border_width, border_colour):
    """
    Draws surface for colour slider, with a full colour gradient as fill colour.

    Args:
        size (list[int, int]): Image size.
        border_width (float): Border width.
        border_colour (pygame.Color): Border colour.

    Returns:
        pygame.Surface: Slider image surface.
    """
    gradient_surface = pygame.Surface(size, pygame.SRCALPHA)

    first_round_end = gradient_surface.height / 2
    second_round_end = gradient_surface.width - first_round_end
    gradient_y_mid = gradient_surface.height / 2

    # Iterate through length of slider
    for i in range(gradient_surface.width):
        draw_height = gradient_surface.height

        if i < first_round_end or i > second_round_end:
            # Draw semicircular caps if x-distance less than or greater than radius of cap (half of image height)
            distance_from_cutoff = min(abs(first_round_end - i), abs(i - second_round_end))
            draw_height = calculate_gradient_slice_height(distance_from_cutoff, gradient_surface.height / 2)

        # Get colour from distance from left side of slider
        color = pygame.Color(0)
        color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)

        draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
        draw_rect.center = (i, gradient_y_mid)

        pygame.draw.rect(gradient_surface, color, draw_rect)

    border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.height))
    pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(border_width), border_radius=int(size[1] / 2))

    return gradient_surface

def calculate_gradient_slice_height(distance, radius):
    """
    Calculate height of vertical slice of semicircular slider cap.

    Args:
        distance (float): x-distance from center of circle.
        radius (float): Radius of semicircle.

    Returns:
        float: Height of vertical slice.
    """
    return sqrt(radius ** 2 - distance ** 2) * 2 + 2

def create_slider_thumb(radius, colour, border_colour, border_width):
    """
    Creates surface with bordered circle.

    Args:
        radius (float): Radius of circle.
        colour (pygame.Color): Fill colour.
        border_colour (pygame.Color): Border colour.
        border_width (float): Border width.

    Returns:
        pygame.Surface: Circle surface.
    """
    thumb_surface = pygame.Surface((radius * 2, radius * 2), pygame.SRCALPHA)
    pygame.draw.circle(thumb_surface, border_colour, (radius, radius), radius, width=int(border_width))
    pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius - border_width))

    return thumb_surface

def create_square_gradient(side_length, colour):
    """
    Creates a square gradient for the colour picker widget, gradient transitioning between saturation and value.
    Uses smoothscale to blend between colour values for individual pixels.

    Args:
        side_length (float): Length of a square side.
        colour (pygame.Color): Colour with desired hue value.

    Returns:
        pygame.Surface: Square gradient surface.
    """
    square_surface = pygame.Surface((side_length, side_length))

    mix_1 = pygame.Surface((1, 2))
    mix_1.fill((255, 255, 255))
    mix_1.set_at((0, 1), (0, 0, 0))
    mix_1 = pygame.transform.smoothscale(mix_1, (side_length, side_length))

    hue = colour.hsva[0]
    saturated_rgb = pygame.Color(0)
    saturated_rgb.hsva = (hue, 100, 100)

    mix_2 = pygame.Surface((2, 1))
    mix_2.fill((255, 255, 255))
    mix_2.set_at((1, 0), saturated_rgb)
    mix_2 = pygame.transform.smoothscale(mix_2,(side_length, side_length))

    mix_1.blit(mix_2, (0, 0), special_flags=pygame.BLEND_MULT)

    square_surface.blit(mix_1, (0, 0))

    return square_surface

def create_switch(size, colour):
    """
    Creates surface for switch toggle widget.

    Args:
        size (list[int, int]): Image size.
        colour (pygame.Color): Fill colour.

    Returns:
        pygame.Surface: Switch surface.
    """
    switch_surface = pygame.Surface((size[0], size[1]), pygame.SRCALPHA)
    pygame.draw.rect(switch_surface, colour, (0, 0, size[0], size[1]), border_radius=int(size[1] / 2))

    return switch_surface

def create_text_box(size, border_width, colours):
    """
    Creates bordered textbox with shadow, flat, and highlighted vertical regions.

    Args:
        size (list[int, int]): Image size.
        border_width (float): Border width.
        colours (list[pygame.Color, ...]): List of 4 colours, representing border colour, shadow colour, flat colour and highlighted colour.

    Returns:
        pygame.Surface: Textbox surface.
    """
    surface = pygame.Surface(size, pygame.SRCALPHA)

    pygame.draw.rect(surface, colours[0], (0, 0, *size))
    pygame.draw.rect(surface, colours[2], (border_width, border_width, size[0] - 2 * border_width, size[1] - 2 * border_width))
    pygame.draw.rect(surface, colours[3], (border_width, border_width, size[0] - 2 * border_width, border_width))
    pygame.draw.rect(surface, colours[1], (border_width, size[1] - 2 * border_width, size[0] - 2 * border_width, border_width))

    return surface
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.chessboard import Chessboard
from data.states.game.components.piece_group import PieceGroup
from data.states.game.components.bitboard_collection import BitboardCollection

class BoardThumbnail(_Widget):
    def __init__(self, relative_width, fen_string='', **kwargs):
        super().__init__(relative_size=(relative_width, relative_width * 0.8), **kwargs)

        self._board = Chessboard(
            parent=self._parent,
            relative_position=(0, 0),
            scale_mode=kwargs.get('scale_mode'),
            relative_width=relative_width
        )

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.initialise_board(fen_string)
        self.set_image()
        self.set_geometry()

    def initialise_board(self, fen_string):
        if len(fen_string) == 0:
            piece_list = []
        else:
            piece_list = BitboardCollection(fen_string).convert_to_piece_list()

        self._piece_group = PieceGroup()
        self._piece_group.initialise_pieces(piece_list, (0, 0), self.size)

        self._board.refresh_board()
        self.set_image()

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)

        self._board.set_image()
        self.image.blit(self._board.image, (0, 0))

        self._piece_group.draw(self.image)

    def set_geometry(self):
        super().set_geometry()
        self._board.set_geometry()

    def set_surface_size(self, new_surface_size):
        super().set_surface_size(new_surface_size)
        self._board.set_surface_size(new_surface_size)
        self._piece_group.handle_resize((0, 0), self.size)

    def process_event(self, event):
        pass
```

```python
import pygame
from data.widgets.bases.pressable import _Pressable
from data.widgets.board_thumbnail import BoardThumbnail
from data.constants import WidgetState
from data.components.custom_event import CustomEvent

class BoardThumbnailButton(_Pressable, BoardThumbnail):
    def __init__(self, event, **kwargs):
        _Pressable.__init__(
            self,
            event=CustomEvent(**vars(event), fen_string=kwargs.get('fen_string')),
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=lambda: self.set_state_colour(WidgetState.BASE),
        )
        BoardThumbnail.__init__(self, **kwargs)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)
```

```python
import pygame
from data.utils.font_helpers import text_width_to_font_size
from data.utils.browser_helpers import get_winner_string
from data.widgets.board_thumbnail import BoardThumbnail
from data.utils.asset_helpers import scale_and_cache
from data.widgets.bases.widget import _Widget

FONT_DIVISION = 7

class BrowserItem(_Widget):
    def __init__(self, relative_width, game, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width * 2), scale_mode='height', **kwargs)

        self._relative_font_size = text_width_to_font_size('YYYY-MM-DD HH:MM:SS', self._font, self.size[0]) / self.surface_size[1]

        self._game = game
        self._board_thumbnail = BoardThumbnail(
            relative_position=(0, 0),
            scale_mode='height',
            relative_width=relative_width,
            fen_string=self._game['final_fen_string']
        )

        self.set_image()
        self.set_geometry()

    def get_text_to_render(self):
        depth_to_text = {
            2: 'EASY',
            3: 'MEDIUM',
            4: 'HARD'
        }

        format_moves = lambda no_of_moves:  int(no_of_moves / 2) if (no_of_moves / 2 % 1 == 0) else round(no_of_moves / 2, 1)

        if self._game['cpu_enabled'] == 1:
            depth_text = depth_to_text[self._game['cpu_depth']]
            cpu_text = f'PVC ({depth_text})'
        else:
            cpu_text = 'PVP'

        return [
            cpu_text,
            self._game['created_dt'].strftime('%Y-%m-%d %H:%M:%S'),
            f'WINNER: {get_winner_string(self._game['winner'])}',
            f'NO. MOVES: {format_moves(self._game['number_of_ply'])}'
        ]

    def set_image(self):
        self.image = pygame.Surface(self.size, pygame.SRCALPHA)
        resized_board = scale_and_cache(self._board_thumbnail.image, (self.size[0], self.size[0] * 0.8))
        self.image.blit(resized_board, (0, 0))

        get_line_y = lambda line: (self.size[0] * 0.8) + ((self.size[0] * 0.8) / FONT_DIVISION) * (line + 0.5)

        text_to_render = self.get_text_to_render()

        for index, text in enumerate(text_to_render):
            self._font.render_to(self.image, (0, get_line_y(index)), text, fgcolor=self._text_colour, size=self.font_size)

    def process_event(self, event):
        pass
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.browser_item import BrowserItem
from data.constants import BrowserEventType
from data.components.custom_event import CustomEvent

WIDTH_FACTOR = 0.3
```

```

class BrowserStrip(_Widget):
    def __init__(self, relative_height, games_list, **kwargs):
        super().__init__(relative_size=None, **kwargs)
        self._relative_item_width = relative_height / 2
        self._get_rect = None

        self._games_list = []
        self._items_list = []
        self._selected_index = None

        self.initialise_games_list(games_list)

    @property
    def item_width(self):
        return self._relative_item_width * self.surface_size[1]

    @property
    def size(self):
        if self._get_rect:
            height = self._get_rect().height
        else:
            height = 0
        width = max(0, len(self._games_list) * (self.item_width + self.margin) + self.margin)

        return (width, height)

    def register_get_rect(self, get_rect_func):
        self._get_rect = get_rect_func

    def initialise_games_list(self, games_list):
        self._items_list = []
        self._games_list = games_list
        self._selected_index = None

        for game in games_list:
            browser_item = BrowserItem(relative_position=(0, 0), game=game, relative_width=self._relative_item_width)
            self._items_list.append(browser_item)

        self.set_image()
        self.set_geometry()

    def set_image(self):
        self.image = pygame.Surface(self.size, pygame.SRCALPHA)
        browser_list = []

        for index, item in enumerate(self._items_list):
            item.set_image()
            browser_list.append((item.image, (index * (self.item_width + self.margin) + self.margin, self.margin)))

        self.image.blits(browser_list)

        if self._selected_index is not None:
            border_position = (self._selected_index * (self.item_width + self.margin), 0)
            border_size = (self.item_width + 2 * self.margin, self.size[1])
            pygame.draw.rect(self.image, (255, 255, 255), (*border_position, *border_size), width=int(self.item_width / 20))

```

```python
    def set_geometry(self):
        super().set_geometry()
        for item in self._items_list:
            item.set_geometry()

    def set_surface_size(self, new_surface_size):
        super().set_surface_size(new_surface_size)

        for item in self._items_list:
            item.set_surface_size(new_surface_size)

    def process_event(self, event, scrolled_pos):
        parent_pos = self._get_rect().topleft
        self.rect.topleft = parent_pos

        if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            self._selected_index = None
            self.set_image()
            return CustomEvent(BrowserEventType.BROWSER_STRIP_CLICK, selected_index=None)

        if event.type == pygame.MOUSEBUTTONDOWN and self.rect.collidepoint(event.pos):
            relative_mouse_pos = (event.pos[0] - parent_pos[0], event.pos[1] - parent_pos[1])
            self._selected_index = int(max(0, (relative_mouse_pos[0] - self.margin) // (self.item_width + self.margin)))
            self.set_image()
            return CustomEvent(BrowserEventType.BROWSER_STRIP_CLICK, selected_index=self._selected_index)
```

```python
import pygame
from data.widgets.reactive_icon_button import ReactiveIconButton
from data.components.custom_event import CustomEvent
from data.widgets.bases.circular import _Circular
from data.widgets.bases.widget import _Widget
from data.constants import Miscellaneous
from data.assets import GRAPHICS, SFX

class Carousel(_Circular, _Widget):
    def __init__(self, event, widgets_dict, **kwargs):
        _Circular.__init__(self, items_dict=widgets_dict)
        _Widget.__init__(self, relative_size=None, **kwargs)

        max_widget_size = (
            max([widget.rect.width for widget in widgets_dict.values()]),
            max([widget.rect.height for widget in widgets_dict.values()])
        )

        self._relative_max_widget_size = (max_widget_size[0] / self.surface_size[1], max_widget_size[1] / self.surface_size[1])
        self._relative_size = ((max_widget_size[0] + 2 * (self.margin + self.arrow_size[0])) / self.surface_size[1], (max_widget_size[1]) / self.surface_size[1])

        self._left_arrow = ReactiveIconButton(
            relative_position=(0, 0),
            relative_size=(0, self.arrow_size[1] / self.surface_size[1]),
            scale_mode='height',
            base_icon=GRAPHICS['left_arrow_base'],
            hover_icon=GRAPHICS['left_arrow_hover'],
            press_icon=GRAPHICS['left_arrow_press'],
            event=CustomEvent(Miscellaneous.PLACEHOLDER),
            sfx=SFX['carousel_click']
        )
        self._right_arrow = ReactiveIconButton(
            relative_position=(0, 0),
            relative_size=(0, self.arrow_size[1] / self.surface_size[1]),
            scale_mode='height',
            base_icon=GRAPHICS['right_arrow_base'],
            hover_icon=GRAPHICS['right_arrow_hover'],
            press_icon=GRAPHICS['right_arrow_press'],
            event=CustomEvent(Miscellaneous.PLACEHOLDER),
            sfx=SFX['carousel_click']
        )

        self._event = event
        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.set_image()
        self.set_geometry()

    @property
    def max_widget_size(self):
        return (self._relative_max_widget_size[0] * self.surface_size[1], self._relative_max_widget_size[1] * self.surface_size[1])

    @property
    def arrow_size(self):
        height = self.max_widget_size[1] * 0.75
        width = (GRAPHICS['left_arrow_base'].width / GRAPHICS['left_arrow_base'].height) * height
        return (width, height)

    @property
    def size(self):
        return ((self.arrow_size[0] + self.margin) * 2 + self.max_widget_size[0], self.max_widget_size[1])

    @property
    def left_arrow_position(self):
        return (0, (self.size[1] - self.arrow_size[1]) / 2)

    @property
    def right_arrow_position(self):
        return (self.size[0] - self.arrow_size[0], (self.size[1] - self.arrow_size[1]) / 2)

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
        self.image.fill(self._fill_colour)

        if self.border_width:
            pygame.draw.rect(self.image, self._border_colour, (0, 0, *self.size), width=int(self.border_width), border_radius=int(self.border_radius))

        self._left_arrow.set_image()
        self.image.blit(self._left_arrow.image, self.left_arrow_position)

        self.current_item.set_image()
        self.image.blit(self.current_item.image, ((self.size[0] - self.current_item.rect.size[0]) / 2, (self.size[1] - self.current_item.rect.size[1]) / 2))

        self._right_arrow.set_image()
        self.image.blit(self._right_arrow.image, self.right_arrow_position)

    def set_geometry(self):
        super().set_geometry()

        self.current_item.set_geometry()
        self._left_arrow.set_geometry()
        self._right_arrow.set_geometry()

        self.current_item.rect.center = self.rect.center
        self._left_arrow.rect.topleft = (self.position[0] + self.
    left_arrow_position[0], self.position[1] + self.left_arrow_position[1])
        self._right_arrow.rect.topleft = (self.position[0] + self.
    right_arrow_position[0], self.position[1] + self.right_arrow_position[1])

    def set_surface_size(self, new_surface_size):
        super().set_surface_size(new_surface_size)
        self._left_arrow.set_surface_size(new_surface_size)
        self._right_arrow.set_surface_size(new_surface_size)

        for item in self._items_dict.values():
            item.set_surface_size(new_surface_size)

    def process_event(self, event):
        self.current_item.process_event(event)
        left_arrow_event = self._left_arrow.process_event(event)
        right_arrow_event = self._right_arrow.process_event(event)

        if left_arrow_event:
            self.set_previous_item()
            self.current_item.set_surface_size(self._raw_surface_size)

        elif right_arrow_event:
            self.set_next_item()
            self.current_item.set_surface_size(self._raw_surface_size)

        if left_arrow_event or right_arrow_event:
            self.set_image()
            self.set_geometry()

            return CustomEvent(**vars(self._event), data=self.current_key)

        elif event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
    MOUSEMOTION]:
            self.set_image()
            self.set_geometry()
```

```
import pygame
from data.widgets.bases.widget import _Widget
from data.utils.board_helpers import create_board
from data.utils.data_helpers import get_user_settings
from data.constants import CursorMode
from data.managers.cursor import cursor

class Chessboard(_Widget):
    def __init__(self, relative_width, change_cursor=True, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width * 0.8), **
    kwargs)

        self._board_surface = None
        self._change_cursor = change_cursor
        self._cursor_is_hand = False

        self.refresh_board()
        self.set_image()
        self.set_geometry()

    def refresh_board(self):
        user_settings = get_user_settings()
        self._board_surface = create_board(self.size, user_settings['primaryBoardColour'], user_settings['secondaryBoardColour'])

        self.set_image()

    def set_image(self):
        self.image = pygame.transform.smoothscale(self._board_surface, self.size)

    def process_event(self, event):
        if self._change_cursor and event.type in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONUP, pygame.MOUSEBUTTONDOWN]:
            current_cursor = cursor.get_mode()

            if self.rect.collidepoint(event.pos):
                if current_cursor == CursorMode.ARROW:
                    cursor.set_mode(CursorMode.OPENHAND)
                elif current_cursor == CursorMode.OPENHAND and (pygame.mouse.get_pressed()[0] is True or event.type == pygame.MOUSEBUTTONDOWN):
                    cursor.set_mode(CursorMode.CLOSEDHAND)
                elif current_cursor == CursorMode.CLOSEDHAND and (pygame.mouse.get_pressed()[0] is False or event.type == pygame.MOUSEBUTTONUP):
                    cursor.set_mode(CursorMode.OPENHAND)
            else:
                if current_cursor == CursorMode.OPENHAND or (current_cursor == CursorMode.CLOSEDHAND and event.type == pygame.MOUSEBUTTONUP):
                    cursor.set_mode(CursorMode.ARROW)
```

```
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.bases.pressable import _Pressable
from data.constants import WidgetState

class ColourButton(_Pressable, _Widget):
    def __init__(self, event, **kwargs):
        _Pressable.__init__(
            self,
            event=event,
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=lambda: self.set_state_colour(WidgetState.BASE),
            sfx=None
        )
        _Widget.__init__(self, **kwargs)

        self._empty_surface = pygame.Surface(self.size)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)

        self.set_image()
        self.set_geometry()

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
```

```python
        self.image.fill(self._fill_colour)
        pygame.draw.rect(self.image, self._border_colour, (0, 0, self.size[0], self.size[1]), width=int(self.border_width))
```

```python
import pygame
from data.widgets.bases.widget import _Widget

class _ColourDisplay(_Widget):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self._colour = None

        self._empty_surface = pygame.Surface(self.size)

    def set_colour(self, new_colour):
        self._colour = new_colour

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
        self.image.fill(self._colour)

    def process_event(self, event):
        pass
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.colour_square import _ColourSquare
from data.widgets.colour_slider import _ColourSlider
from data.widgets.colour_display import _ColourDisplay
from data.components.custom_event import CustomEvent

class ColourPicker(_Widget):
    def __init__(self, relative_width, event_type, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width), scale_mode='width', **kwargs)

        self.image = pygame.Surface(self.size)
        self.rect = self.image.get_rect()

        self._square = _ColourSquare(
            parent=self,
            relative_position=(0.1, 0.1),
            relative_width=0.5,
            event_type=event_type
        )
        self._square.set_colour(kwargs.get('selected_colour'))

        self._slider = _ColourSlider(
            parent=self,
            relative_position=(0.0, 0.7),
            relative_width=1.0,
            border_width=self.border_width,
            border_colour=self._border_colour
        )
        self._slider.set_colour(kwargs.get('selected_colour'))

        self._display = _ColourDisplay(
            parent=self,
            relative_position=(0.7, 0.1),
            relative_size=(0.2, 0.5)
        )
        self._display.set_colour(kwargs.get('selected_colour'))

        self._event_type = event_type
        self._hover_event_type = event_type

        self.set_image()
        self.set_geometry()

    def global_to_relative_pos(self, global_pos):
        return (global_pos[0] - self.position[0], global_pos[1] - self.position[1])

    def set_image(self):
        self.image = pygame.Surface(self.size)
        self.image.fill(self._fill_colour)

        self._square.set_image()
        self._square.set_geometry()
        self.image.blit(self._square.image, self.global_to_relative_pos(self._square.position))

        self._slider.set_image()
        self._slider.set_geometry()
        self.image.blit(self._slider.image, self.global_to_relative_pos(self._slider.position))

        self._display.set_image()
        self._display.set_geometry()
        self.image.blit(self._display.image, self.global_to_relative_pos(self._display.position))

        pygame.draw.rect(self.image, self._border_colour, (0, 0, self.size[0], self.size[1]), width=int(self.border_width))

    def set_surface_size(self, new_surface_size):
        super().set_surface_size(new_surface_size)
        self._square.set_surface_size(self.size)
        self._slider.set_surface_size(self.size)
        self._display.set_surface_size(self.size)

    def get_picker_position(self):
        return self.position

    def process_event(self, event):
        slider_colour = self._slider.process_event(event)
        square_colour = self._square.process_event(event)

        if square_colour:
            self._display.set_colour(square_colour)
            self.set_image()

        if slider_colour:
            self._square.set_colour(slider_colour)
            self.set_image()

        if event.type in [pygame.MOUSEBUTTONUP, pygame.MOUSEBUTTONDOWN, pygame.MOUSEMOTION] and self.rect.collidepoint(event.pos):
            return CustomEvent(self._event_type, colour=square_colour)
```

```
import pygame
from data.utils.widget_helpers import create_slider_gradient
from data.utils.asset_helpers import smoothscale_and_cache
```

```python
from data.widgets.slider_thumb import _SliderThumb
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState

class _ColourSlider(_Widget):
    def __init__(self, relative_width, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width * 0.2), **kwargs)

        # Initialise slider thumb.
        self._thumb = _SliderThumb(radius=self.size[1] / 2, border_colour=self._border_colour)

        self._selected_percent = 0
        self._last_mouse_x = None

        self._gradient_surface = create_slider_gradient(self.gradient_size, self.border_width, self._border_colour)
        self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)

    @property
    def gradient_size(self):
        return (self.size[0] - 2 * (self.size[1] / 2), self.size[1] / 2)

    @property
    def gradient_position(self):
        return (self.size[1] / 2, self.size[1] / 4)

    @property
    def thumb_position(self):
        return (self.gradient_size[0] * self._selected_percent, 0)

    @property
    def selected_colour(self):
        colour = pygame.Color(0)
        colour.hsva = (int(self._selected_percent * 360), 100, 100, 100)
        return colour

    def calculate_gradient_percent(self, mouse_pos):
        """
        Calculate what percentage slider thumb is at based on change in mouse position.

        Args:
            mouse_pos (list[int, int]): Position of mouse on window screen.

        Returns:
            float: Slider scroll percentage.
        """
        if self._last_mouse_x is None:
            return

        x_change = (mouse_pos[0] - self._last_mouse_x) / (self.gradient_size[0] - 2 * self.border_width)
        return max(0, min(self._selected_percent + x_change, 1))

    def relative_to_global_position(self, position):
        """
        Transforms position from being relative to widget rect, to window screen.

        Args:
            position (list[int, int]): Position relative to widget rect.

        Returns:
            list[int, int]: Position relative to window screen.
        """
        relative_x, relative_y = position
        return (relative_x + self.position[0], relative_y + self.position[1])

    def set_colour(self, new_colour):
        """
        Sets selected_percent based on the new colour's hue.

        Args:
            new_colour (pygame.Color): New slider colour.
        """
        colour = pygame.Color(new_colour)
        hue = colour.hsva[0]
        self._selected_percent = hue / 360
        self.set_image()

    def set_image(self):
        """
        Draws colour slider to widget image.
        """
        # Scales initalised gradient surface instead of redrawing it everytime set_image is called
        gradient_scaled = smoothscale_and_cache(self._gradient_surface, self.gradient_size)

        self.image = pygame.transform.scale(self._empty_surface, (self.size))
        self.image.blit(gradient_scaled, self.gradient_position)

        # Resets thumb colour, image and position, then draws it to the widget image
        self._thumb.initialise_new_colours(self.selected_colour)
        self._thumb.set_surface(radius=self.size[1] / 2, border_width=self.border_width)
        self._thumb.set_position(self.relative_to_global_position((self.thumb_position[0], self.thumb_position[1])))

        thumb_surface = self._thumb.get_surface()
        self.image.blit(thumb_surface, self.thumb_position)

    def process_event(self, event):
        """
        Processes Pygame events.

        Args:
            event (pygame.Event): Event to process.

        Returns:
            pygame.Color: Current colour slider is displaying.
        """
        if event.type not in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP]:
            return

        # Gets widget state before and after event is processed by slider thumb
        before_state = self._thumb.state
        self._thumb.process_event(event)
        after_state = self._thumb.state

        # If widget state changes (e.g. hovered -> pressed), redraw widget
        if before_state != after_state:
            self.set_image()

        if event.type == pygame.MOUSEMOTION:
            if self._thumb.state == WidgetState.PRESS:
                # Recalculates slider colour based on mouse position change
                selected_percent = self.calculate_gradient_percent(event.pos)
                self._last_mouse_x = event.pos[0]

                if selected_percent is not None:
                    self._selected_percent = selected_percent

                    return self.selected_colour

        if event.type == pygame.MOUSEBUTTONUP:
            # When user stops scrolling, return new slider colour
            self._last_mouse_x = None
            return self.selected_colour

        if event.type == pygame.MOUSEBUTTONDOWN or before_state != after_state:
            # Redraws widget when slider thumb is hovered or pressed
            return self.selected_colour
```

```
import pygame
from data.widgets.bases.widget import _Widget
from data.utils.widget_helpers import create_square_gradient

class _ColourSquare(_Widget):
    def __init__(self, relative_width, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width), scale_mode='width', **kwargs)

        self._colour = None

    def set_colour(self, new_colour):
        self._colour = pygame.Color(new_colour)

    def get_colour(self):
        return self._colour

    def set_image(self):
        self.image = create_square_gradient(side_length=self.size[0], colour=self._colour)

    def process_event(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            relative_mouse_pos = (event.pos[0] - self.position[0], event.pos[1] - self.position[1])

            if (
                0 > relative_mouse_pos[0] or
                self.size[0] < relative_mouse_pos[0] or
                0 > relative_mouse_pos[1] or
                self.size[1] < relative_mouse_pos[1]
            ): return None

            self.set_colour(self.image.get_at(relative_mouse_pos))

            return self._colour

        return None
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.bases.pressable import _Pressable
from data.constants import WidgetState
from data.utils.data_helpers import get_user_settings
from data.utils.font_helpers import text_width_to_font_size, text_height_to_font_size
from data.assets import GRAPHICS, FONTS

user_settings = get_user_settings()

class Dropdown(_Pressable, _Widget):
    def __init__(self, word_list, event=None, **kwargs):
        _Pressable.__init__(
            self,
            event=event,
            hover_func=self.hover_func,
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=self.up_func,
            sfx=None
        )
        _Widget.__init__(self, relative_size=None, **kwargs)

        if kwargs.get('relative_width'):
            self._relative_font_size = text_width_to_font_size(max(word_list, key=len), self._font, kwargs.get('relative_width') * self.surface_size[0] - self.margin) / self.surface_size[1]
        elif kwargs.get('relative_height'):
            self._relative_font_size = text_height_to_font_size(max(word_list, key=len), self._font, kwargs.get('relative_height') * self.surface_size[1] - self.margin) / self.surface_size[1]

        self._word_list = [word_list[0].capitalize()]
        self._word_list_copy = [word.capitalize() for word in word_list]

        self._expanded = False
        self._hovered_index = None

        self._empty_surface = pygame.Surface((0, 0))
        self._background_colour = self._fill_colour

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)

        self.set_image()
        self.set_geometry()

    @property
    def size(self):
        max_word = sorted(self._word_list_copy, key=len)[-1]
        max_word_rect = self._font.get_rect(max_word, size=self.font_size)
        all_words_rect = pygame.FRect(0, 0, max_word_rect.size[0], (max_word_rect.size[1] * len(self._word_list)) + (self.margin * (len(self._word_list) - 1)))
        all_words_rect = all_words_rect.inflate(2 * self.margin, 2 * self.margin)
        return (all_words_rect.size[0] + max_word_rect.size[1], all_words_rect.size[1])

    def get_selected_word(self):
        return self._word_list[0].lower()

    def toggle_expanded(self):
        if self._expanded:
            self._word_list = [self._word_list_copy[0]]
        else:
            self._word_list = [*self._word_list_copy]

        self._expanded = not(self._expanded)

    def hover_func(self):
        mouse_position = pygame.mouse.get_pos()
        relative_position = (mouse_position[0] - self.position[0], mouse_position[1] - self.position[1])
        self._hovered_index = self.calculate_hovered_index(relative_position)
        self.set_state_colour(WidgetState.HOVER)

    def set_selected_word(self, word):
        index = self._word_list_copy.index(word.capitalize())
        selected_word = self._word_list_copy.pop(index)
        self._word_list_copy.insert(0, selected_word)

        if self._expanded:
            self._word_list.pop(index)
            self._word_list.insert(0, selected_word)
        else:
            self._word_list = [selected_word]

        self.set_image()

    def up_func(self):
        if self.get_widget_state() == WidgetState.PRESS:
            if self._expanded and self._hovered_index is not None:
                self.set_selected_word(self._word_list_copy[self._hovered_index])

            self.toggle_expanded()

        self._hovered_index = None

        self.set_state_colour(WidgetState.BASE)
        self.set_geometry()

    def calculate_hovered_index(self, mouse_pos):
        return int(mouse_pos[1] // (self.size[1] / len(self._word_list)))

    def set_image(self):
        text_surface = pygame.transform.scale(self._empty_surface, self.size)
        self.image = text_surface

        fill_rect = pygame.FRect(0, 0, self.size[0], self.size[1])
        pygame.draw.rect(self.image, self._background_colour, fill_rect)
        pygame.draw.rect(self.image, self._border_colour, fill_rect, width=int(self.border_width))

        word_box_height = (self.size[1] - (2 * self.margin) - ((len(self._word_list) - 1) * self.margin)) / len(self._word_list)

        arrow_size = (GRAPHICS['dropdown_arrow_open'].width / GRAPHICS['dropdown_arrow_open'].height * word_box_height, word_box_height)
        open_arrow_surface = pygame.transform.scale(GRAPHICS['dropdown_arrow_open'], arrow_size)
        closed_arrow_surface = pygame.transform.scale(GRAPHICS['dropdown_arrow_close'], arrow_size)
        arrow_position = (self.size[0] - arrow_size[0] - self.margin, (word_box_height) / 3)

```

```python
        if self._expanded:
            self.image.blit(closed_arrow_surface, arrow_position)
        else:
            self.image.blit(open_arrow_surface, arrow_position)

        for index, word in enumerate(self._word_list):
            word_position = (self.margin, self.margin + (word_box_height + self.margin) * index)
            self._font.render_to(self.image, word_position, word, fgcolor=self._text_colour, size=self.font_size)

        if self._hovered_index is not None:
            overlay_surface = pygame.Surface((self.size[0], word_box_height + 2 * self.margin), pygame.SRCALPHA)
            overlay_surface.fill((*self._fill_colour.rgb, 128))
            overlay_position = (0, (word_box_height + self.margin) * self._hovered_index)
            self.image.blit(overlay_surface, overlay_position)
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.utils.widget_helpers import create_text_box

class Icon(_Widget):
    def __init__(self, icon, stretch=False, is_mask=False, smooth=False, fit_icon=False, box_colours=None, **kwargs):
        super().__init__(**kwargs)

        if fit_icon:
            aspect_ratio = icon.width / icon.height
            self._relative_size = (self._relative_size[1] * aspect_ratio, self._relative_size[1])

        self._icon = icon
        self._is_mask = is_mask
        self._stretch = stretch
        self._smooth = smooth
        self._box_colours = box_colours

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.set_image()
        self.set_geometry()

    def set_icon(self, icon):
        self._icon = icon
        self.set_image()

    def set_image(self):
        if self._box_colours:
            self.image = create_text_box(self.size, self.border_width, self._box_colours)
        else:
            self.image = pygame.transform.scale(self._empty_surface, self.size)

            if self._fill_colour:
                pygame.draw.rect(self.image, self._fill_colour, self.image.get_rect(), border_radius=int(self.border_radius))

        if self._stretch:
            if self._smooth:
                scaled_icon = pygame.transform.smoothscale(self._icon, (self.size
[0] - (2 * self.margin), self.size[1] - (2 * self.margin)))
            else:
                scaled_icon = pygame.transform.scale(self._icon, (self.size[0] - (2 * self.margin), self.size[1] - (2 * self.margin)))

            icon_position = (self.margin, self.margin)
        else:
            max_height = self.size[1] - (2 * self.margin)
            max_width = self.size[0] - (2 * self.margin)
            scale_factor = min(max_width / self._icon.width, max_height / self._icon.height)

            if self._smooth:
                scaled_icon = pygame.transform.smoothscale_by(self._icon, (scale_factor, scale_factor))
            else:
                scaled_icon = pygame.transform.scale_by(self._icon, (scale_factor, scale_factor))
            icon_position = ((self.size[0] - scaled_icon.width) / 2, (self.size[1] - scaled_icon.height) / 2)

        if self._is_mask:
            self.image.blit(scaled_icon, icon_position, None, pygame.BLEND_RGBA_MULT)
        else:
            self.image.blit(scaled_icon, icon_position)

        if self._box_colours is None and self.border_width:
            pygame.draw.rect(self.image, self._border_colour, self.image.get_rect(), width=int(self.border_width), border_radius=int(self.border_radius))

    def process_event(self, event):
        pass
```

```
from data.widgets.bases.pressable import _Pressable
from data.widgets.bases.box import _Box
from data.widgets.icon import Icon
from data.constants import WidgetState, RED_BUTTON_COLOURS

class IconButton(_Box, _Pressable, Icon):
    def __init__(self, event, box_colours=RED_BUTTON_COLOURS, **kwargs):
        _Box.__init__(self, box_colours=box_colours)
        _Pressable.__init__(
            self,
            event=event,
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=lambda: self.set_state_colour(WidgetState.BASE),
        )
        Icon.__init__(self, box_colours=box_colours[WidgetState.BASE], **kwargs)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)
```

```
import pygame
from data.widgets.bases.widget import _Widget
from data.utils.font_helpers import width_to_font_size

class MoveList(_Widget):
    def __init__(self, relative_width, minimum_height=0, move_list=[], **kwargs):
        super().__init__(relative_size=None, **kwargs)

        self._relative_width = relative_width * self.surface_size[0] / self.surface_size[1]
        self._relative_minimum_height = minimum_height / self.surface_size[1]
        self._move_list = move_list
        self._relative_font_size = width_to_font_size(self._font, self.surface_size[0] / 3.5) / self.surface_size[1]

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.set_image()
        self.set_geometry()

    @property
    def size(self):
        font_metrics = self._font.get_metrics('j', size=self.font_size)

        width = self._relative_width * self.surface_size[1]
        minimum_height = self._relative_minimum_height * self.surface_size[1]
        row_gap = font_metrics[0][3] - font_metrics[0][2]
        number_of_rows = 2 * ((len(self._move_list) + 1) // 2) + 1

        return (width, max(minimum_height, row_gap * number_of_rows))

    def register_get_rect(self, get_rect_func):
        pass

    def reset_move_list(self):
        self._move_list = []
        self.set_image()
        self.set_geometry()

    def append_to_move_list(self, new_move):
        self._move_list.append(new_move)
        self.set_image()
        self.set_geometry()

    def pop_from_move_list(self):
        self._move_list.pop()
        self.set_image()
        self.set_geometry()

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
        self.image.fill(self._fill_colour)

        font_metrics = self._font.get_metrics('j', size=self.font_size)
        row_gap = font_metrics[0][3] - font_metrics[0][2]

        for index, move in enumerate(self._move_list):
            if index % 2 == 0:
                text_position = (self.size[0] / 7, row_gap * (1 + 2 * (index // 2)))
            else:
                text_position = (self.size[0] * 4 / 7, row_gap * (1 + 2 * (index // 2)))

            self._font.render_to(self.image, text_position, text=move, size=self.font_size, fgcolor=self._text_colour)

            move_number = (index // 2) + 1
            move_number_position = (self.size[0] / 14, row_gap * (1 + 2 * (index // 2)))
```

```python
            self._font.render_to(self.image, move_number_position, text=str(move_number), size=self.font_size, fgcolor=self._text_colour)

    def process_event(self, event, scrolled_pos=None):
        pass
```

```python
import pygame
from data.constants import WidgetState, LOCKED_BLUE_BUTTON_COLOURS, LOCKED_RED_BUTTON_COLOURS, RED_BUTTON_COLOURS, BLUE_BUTTON_COLOURS
from data.components.custom_event import CustomEvent
from data.widgets.bases.circular import _Circular
from data.widgets.icon_button import IconButton
from data.widgets.bases.box import _Box

class MultipleIconButton(_Circular, IconButton):
  def __init__(self, icons_dict, **kwargs):
    _Circular.__init__(self, items_dict=icons_dict)
    IconButton.__init__(self, icon=self.current_item, **kwargs)

    self._fill_colour_copy = self._fill_colour

    self._locked = None

  def set_locked(self, is_locked):
    self._locked = is_locked
    if self._locked:
      r, g, b, a  = pygame.Color(self._fill_colour_copy).rgba
      if self._box_colours_dict == BLUE_BUTTON_COLOURS:
        _Box.__init__(self, box_colours=LOCKED_BLUE_BUTTON_COLOURS)
      elif self._box_colours_dict == RED_BUTTON_COLOURS:
        _Box.__init__(self, box_colours=LOCKED_RED_BUTTON_COLOURS)
      else:
        self.initialise_new_colours((max(r + 50, 0), max(g + 50, 0), max(b + 50, 0), a))
    else:
      if self._box_colours_dict == LOCKED_BLUE_BUTTON_COLOURS:
        _Box.__init__(self, box_colours=BLUE_BUTTON_COLOURS)
      elif self._box_colours_dict == LOCKED_RED_BUTTON_COLOURS:
        _Box.__init__(self, box_colours=RED_BUTTON_COLOURS)
      else:
        self.initialise_new_colours(self._fill_colour_copy)

    if self.rect.collidepoint(pygame.mouse.get_pos()):
      self.set_state_colour(WidgetState.HOVER)
    else:
      self.set_state_colour(WidgetState.BASE)

  def set_next_icon(self):
    super().set_next_item()
    self._icon = self.current_item
    self.set_image()

  def process_event(self, event):
    widget_event = super().process_event(event)

    if widget_event:
      return CustomEvent(**vars(widget_event), data=self.current_key)
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.states.game.components.piece_sprite import PieceSprite
from data.constants import Score, Rotation, WidgetState, Colour, BLUE_BUTTON_COLOURS, RED_BUTTON_COLOURS
from data.utils.widget_helpers import create_text_box
from data.utils.asset_helpers import scale_and_cache

class PieceDisplay(_Widget):
    def __init__(self, active_colour, **kwargs):
        super().__init__(**kwargs)

        self._active_colour = active_colour
        self._piece_list = []
        self._piece_surface = None
        self._box_colours = BLUE_BUTTON_COLOURS[WidgetState.BASE] if active_colour == Colour.BLUE else RED_BUTTON_COLOURS[WidgetState.BASE]

        self.initialise_piece_surface()

        self.set_image()
        self.set_geometry()

    def add_piece(self, piece):
        self._piece_list.append(piece)
        self._piece_list.sort(key=lambda piece: Score[piece.name])
        self.initialise_piece_surface()

    def remove_piece(self, piece):
        self._piece_list.remove(piece)
        self.initialise_piece_surface()

    def reset_piece_list(self):
        self._piece_list = []
        self.initialise_piece_surface()

    def initialise_piece_surface(self):
        self._piece_surface = pygame.Surface((self.size[0] - 2 * self.margin, self.size[1] - 2 * self.margin), pygame.SRCALPHA)

        if (len(self._piece_list) == 0):
            self.set_image()
            return

        piece_width = min(self.size[1] - 2 * self.margin, (self.size[0] - 2 * self.margin) / len(self._piece_list))
        piece_list = []

        for index, piece in enumerate(self._piece_list):
            piece_instance = PieceSprite(piece, self._active_colour.get_flipped_colour(), Rotation.UP)
            piece_instance.set_geometry((0, 0), piece_width)
            piece_instance.set_image()
            piece_list.append((piece_instance.image, (piece_width * index, (self._piece_surface.height - piece_width) / 2)))

        self._piece_surface.fblits(piece_list)

        self.set_image()

    def set_image(self):
        self.image = create_text_box(self.size, self.border_width, self._box_colours)

        resized_piece_surface = scale_and_cache(self._piece_surface, (self.size[0]
 - 2 * self.margin, self.size[1] - 2 * self.margin))
        self.image.blit(resized_piece_surface, (self.margin, self.margin))

    def process_event(self, event):
        pass
```

```
from data.components.custom_event import CustomEvent
from data.widgets.bases.pressable import _Pressable
from data.widgets.bases.circular import _Circular
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState

class ReactiveButton(_Pressable, _Circular, _Widget):
    def __init__(self, widgets_dict, event, center=False, **kwargs):
        # Multiple inheritance used here, to combine the functionality of multiple
 super classes
        _Pressable.__init__(
            self,
            event=event,
            hover_func=lambda: self.set_to_key(WidgetState.HOVER),
            down_func=lambda: self.set_to_key(WidgetState.PRESS),
            up_func=lambda: self.set_to_key(WidgetState.BASE),
            **kwargs
        )
        # Aggregation used to cycle between external widgets
        _Circular.__init__(self, items_dict=widgets_dict)
        _Widget.__init__(self, **kwargs)

        self._center = center

        self.initialise_new_colours(self._fill_colour)

    @property
    def position(self):
        """
        Overrides position getter method, to always position icon in the center if
 self._center is True.

        Returns:
            list[int, int]: Position of widget.
        """
        position = super().position

        if self._center:
            self._size_diff = (self.size[0] - self.rect.width, self.size[1] - self
.rect.height)
            return (position[0] + self._size_diff[0] / 2, position[1] + self.
_size_diff[1] / 2)
        else:
            return position

    def set_image(self):
        """
        Sets current icon to image.
        """
        self.current_item.set_image()
        self.image = self.current_item.image

    def set_geometry(self):
        """
        Sets size and position of widget.
        """
```

```python
        super().set_geometry()
        self.current_item.set_geometry()
        self.current_item.rect.topleft = self.rect.topleft

    def set_surface_size(self, new_surface_size):
        """
        Overrides base method to resize every widget state icon, not just the current one.

        Args:
            new_surface_size (list[int, int]): New surface size.
        """
        super().set_surface_size(new_surface_size)
        for item in self._items_dict.values():
            item.set_surface_size(new_surface_size)

    def process_event(self, event):
        """
        Processes Pygame events.

        Args:
            event (pygame.Event): Event to process.

        Returns:
            CustomEvent: CustomEvent of current item, with current key included
        """
        widget_event = super().process_event(event)
        self.current_item.process_event(event)

        if widget_event:
            return CustomEvent(**vars(widget_event), data=self.current_key)
```

```python
from data.widgets.reactive_button import ReactiveButton
from data.constants import WidgetState
from data.widgets.icon import Icon

class ReactiveIconButton(ReactiveButton):
    def __init__(self, base_icon, hover_icon, press_icon, **kwargs):
        # Composition is used here, to initialise the Icon widgets for each widget state
        widgets_dict = {
            WidgetState.BASE: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=base_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.HOVER: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=hover_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.PRESS: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=press_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            )
        }

        super().__init__(
            widgets_dict=widgets_dict,
            **kwargs
        )
```

```python
import pygame
from data.widgets.bases.widget import _Widget

class Rectangle(_Widget):
    def __init__(self, visible=False, **kwargs):
        super().__init__(**kwargs)

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)
        self._visible = visible

        self.set_image()
        self.set_geometry()

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
        if self._visible:
            pygame.draw.rect(self.image, self._fill_colour, self.image.get_rect(), border_radius=int(self.border_radius))

            if self.border_width:
                pygame.draw.rect(self.image, self._border_colour, self.image.get_rect(), width=int(self.border_width), border_radius=int(self.border_radius))

    def process_event(self, event):
        pass
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.bases.pressable import _Pressable
from data.constants import WidgetState, Miscellaneous

# self.set_state_colour(WidgetState.HOVER)
class _Scrollbar(_Pressable, _Widget):
    def __init__(self, vertical, **kwargs):
        _Pressable.__init__(
            self,
            event=Miscellaneous.PLACEHOLDER,
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=self.down_func,
            up_func=self.up_func,
            prolonged=True,
            sfx=None
        )
        _Widget.__init__(self, **kwargs)

        self._vertical = vertical
        self._last_mouse_px = None

        self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)

        self.set_image()
        self.set_geometry()

    def down_func(self):
        if self._vertical:
            self._last_mouse_px = pygame.mouse.get_pos()[1]
        else:
            self._last_mouse_px = pygame.mouse.get_pos()[0]

        self.set_state_colour(WidgetState.PRESS)

    def up_func(self):
        self._last_mouse_px = None
        self.set_state_colour(WidgetState.BASE)

    def set_relative_position(self, relative_position):
        self._relative_position = relative_position
        self.set_geometry()

    def set_relative_size(self, new_relative_size):
        self._relative_size = new_relative_size

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)

        if self._vertical:
            rounded_radius = self.size[0] / 2
        else:
            rounded_radius = self.size[1] / 2

        pygame.draw.rect(self.image, self._fill_colour, (0, 0, self.size[0], self.size[1]), border_radius=int(rounded_radius))

    def process_event(self, event):
        before_state = self.get_widget_state()
        widget_event = super().process_event(event)
        after_state = self.get_widget_state()

        if event.type == pygame.MOUSEMOTION and self._last_mouse_px:
            if self._vertical:
                offset_from_last_frame = event.pos[1] - self._last_mouse_px
                self._last_mouse_px = event.pos[1]

                return offset_from_last_frame
            else:
                offset_from_last_frame = event.pos[0] - self._last_mouse_px
                self._last_mouse_px = event.pos[0]

                return offset_from_last_frame


        if widget_event or before_state != after_state:
            return 0
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.scrollbar import _Scrollbar
from data.managers.theme import theme

SCROLLBAR_WIDTH_FACTOR = 0.05

class ScrollArea(_Widget):
    def __init__(self, widget, vertical, scroll_factor=15, **kwargs):
        super().__init__(**kwargs)
        if vertical is False:
            self._relative_size = kwargs.get('relative_size')

        self._relative_scroll_factor = scroll_factor / self.surface_size[1]

        self._scroll_percentage = 0
        self._widget = widget
        self._vertical = vertical

        self._widget.register_get_rect(self.calculate_widget_rect)

        if self._vertical:
            anchor_x = 'right'
            anchor_y = 'top'
            scale_mode = 'height'
        else:
            anchor_x = 'left'
            anchor_y = 'bottom'
            scale_mode = 'width'

        self._scrollbar = _Scrollbar(
            parent=self,
            relative_position=(0, 0),
            relative_size=None,
            anchor_x=anchor_x,
            anchor_y=anchor_y,
            fill_colour=theme['borderPrimary'],
            scale_mode=scale_mode,
            vertical=vertical,
        )

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.set_image()
        self.set_geometry()

    @property
    def scroll_factor(self):
        return self._relative_scroll_factor * self.surface_size[1]

    @property
    def scrollbar_size(self):
        if self._vertical:
            return (self.size[0] * SCROLLBAR_WIDTH_FACTOR, min(1, self.size[1] /
    self._widget.rect.height) * self.size[1])
        else:
            return (min(1, self.size[0] / (self._widget.rect.width + 0.001)) *
    self.size[0], self.size[1] * SCROLLBAR_WIDTH_FACTOR)

    @property
    def size(self):
        if self._vertical is False:
            return (self._relative_size[0] * self.surface_size[0], self._relative_size[1] * self.surface_size[1]) # scale with horizontal width to always fill entire screen
        else:
            return super().size

    def calculate_scroll_percentage(self, offset, scrollbar=False):
        if self._vertical:
            widget_height = self._widget.rect.height

            if widget_height < self.size[1]:
                return 0

            if scrollbar:
                self._scroll_percentage += offset / (self.size[1] - self.scrollbar_size[1] + 0.001)
            else:
                max_scroll_height = widget_height - self.size[1]
                current_scroll_height = self._scroll_percentage * max_scroll_height
                self._scroll_percentage = (current_scroll_height + offset) / (max_scroll_height + 0.001)
        else:
            widget_width = self._widget.rect.width

            if widget_width < self.size[0]:
                return 0

            if scrollbar:
                self._scroll_percentage += offset / (self.size[0] - self.scrollbar_size[0] + 0.001)
            else:
                max_scoll_width = widget_width - self.size[0]
                current_scroll_width = self._scroll_percentage * max_scoll_width
                self._scroll_percentage = (current_scroll_width + offset) / max_scoll_width

        return min(1, max(0, self._scroll_percentage))

    def calculate_widget_rect(self):
        widget_position = self.calculate_widget_position()
        return pygame.FRect(widget_position[0] - self.position[0], self.position[1] + widget_position[1], self.size[0], self.size[1])

    def calculate_widget_position(self):
        if self._vertical:
            return (0, -self._scroll_percentage * (self._widget.rect.height - self.size[1]))
        else:
            return (-self._scroll_percentage * (self._widget.rect.width - self.size[0]), 0)

    def calculate_relative_scrollbar_position(self):
        if self._vertical:
            vertical_offset = (self.size[1] - self.scrollbar_size[1]) * self._scroll_percentage
            scrollbar_position = (0, vertical_offset)
        else:
            horizontal_offset = (self.size[0] - self.scrollbar_size[0]) * self._scroll_percentage
            scrollbar_position = (horizontal_offset, 0)

```

```python
        return (scrollbar_position[0] / self.size[0], scrollbar_position[1] / self.size[1])

    def set_widget(self, new_widget):
        self._widget = new_widget
        self.set_image()
        self.set_geometry()

    def set_image(self):
        self.image = pygame.transform.scale(self._empty_surface, self.size)
        self.image.fill(theme['fillPrimary'])

        self._widget.set_image()
        self.image.blit(self._widget.image, self.calculate_widget_position())

        self._scrollbar.set_relative_position(self.calculate_relative_scrollbar_position()) # WRONG USING RELATIVE
        self._scrollbar.set_relative_size((self.scrollbar_size[0] / self.size[1], self.scrollbar_size[1] / self.size[1]))
        self._scrollbar.set_image()
        relative_scrollbar_position = (self._scrollbar.rect.left - self.position[0], self._scrollbar.rect.top - self.position[1])
        self.image.blit(self._scrollbar.image, relative_scrollbar_position)

    def set_geometry(self):
        super().set_geometry()
        self._widget.set_geometry()
        self._scrollbar.set_geometry()

    def set_surface_size(self, new_surface_size):
        super().set_surface_size(new_surface_size)
        self._widget.set_surface_size(new_surface_size)
        self._scrollbar.set_surface_size(new_surface_size)

    def process_event(self, event):
        # WAITING FOR PYGAME-CE 2.5.3 TO RELEASE TO FIX SCROLL FLAGS
        # self.image.scroll(0, SCROLL_FACTOR)
        # self.image.scroll(0, -SCROLL_FACTOR)

        offset = self._scrollbar.process_event(event)

        if offset is not None:
            self.set_image()

            if abs(offset) > 0:
                self._scroll_percentage = self.calculate_scroll_percentage(offset, scrollbar=True)

        if self.rect.collidepoint(pygame.mouse.get_pos()):
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 4:
                    self._scroll_percentage = self.calculate_scroll_percentage(-self.scroll_factor)
                    self.set_image()
                    return
                elif event.button == 5:
                    if self._scroll_percentage == 100:
                        return

                    self._scroll_percentage = self.calculate_scroll_percentage(self.scroll_factor)
                    self.set_image()
                    return

        widget_event = self._widget.process_event(event, scrolled_pos=self.calculate_widget_position())
        if widget_event is not None:
            self.set_image()
        return widget_event
```

```python
import pygame
from data.widgets.bases.pressable import _Pressable
from data.constants import WidgetState
from data.utils.widget_helpers import create_slider_thumb
from data.managers.theme import theme

class _SliderThumb(_Pressable):
    def __init__(self, radius, border_colour=theme['borderPrimary'], fill_colour=theme['fillPrimary']):
        super().__init__(
            event=None,
            down_func=self.down_func,
            up_func=self.up_func,
            hover_func=self.hover_func,
            prolonged=True,
            sfx=None
        )
        self._border_colour = border_colour
        self._radius = radius
        self._percent = None

        self.state = WidgetState.BASE
        self.initialise_new_colours(fill_colour)

    def get_position(self):
        return (self.rect.x, self.rect.y)

    def set_position(self, position):
        self.rect = self._thumb_surface.get_rect()
        self.rect.topleft = position

    def get_surface(self):
        return self._thumb_surface

    def set_surface(self, radius, border_width):
        self._thumb_surface = create_slider_thumb(radius, self._colours[self.state], self._border_colour, border_width)

    def get_pressed(self):
        return self._pressed

    def down_func(self):
        self.state = WidgetState.PRESS

    def up_func(self):
        self.state = WidgetState.BASE

    def hover_func(self):
        self.state = WidgetState.HOVER
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.bases.pressable import _Pressable
from data.constants import WidgetState
from data.utils.widget_helpers import create_switch
from data.components.custom_event import CustomEvent
from data.managers.theme import theme

class Switch(_Pressable, _Widget):
    def __init__(self, relative_height, event, fill_colour=theme['fillTertiary'], on_colour=theme['fillSecondary'], off_colour=theme['fillPrimary'], **kwargs):
        _Pressable.__init__(
            self,
            event=event,
            hover_func=self.hover_func,
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=self.up_func,
        )
        _Widget.__init__(self, relative_size=(relative_height * 2, relative_height), scale_mode='height',fill_colour=fill_colour, **kwargs)

        self._on_colour = on_colour
        self._off_colour = off_colour
        self._background_colour = None

        self._is_toggled = None
        self.set_toggle_state(False)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)

        self.set_image()
        self.set_geometry()

    def hover_func(self):
        self.set_state_colour(WidgetState.HOVER)

    def set_toggle_state(self, is_toggled):
        self._is_toggled = is_toggled
        if is_toggled:
            self._background_colour = self._on_colour
        else:
            self._background_colour = self._off_colour

        self.set_image()

    def up_func(self):
        if self.get_widget_state() == WidgetState.PRESS:
            toggle_state = not(self._is_toggled)
            self.set_toggle_state(toggle_state)

        self.set_state_colour(WidgetState.BASE)

    def draw_thumb(self):
        margin = self.size[1] * 0.1
        thumb_radius = (self.size[1] / 2) - margin

        if self._is_toggled:
            thumb_center = (self.size[0] - margin - thumb_radius, self.size[1] / 2)
        else:
            thumb_center = (margin + thumb_radius, self.size[1] / 2)

        pygame.draw.circle(self.image, self._fill_colour, thumb_center, thumb_radius)

    def set_image(self):
        self.image = create_switch(self.size, self._background_colour)
        self.draw_thumb()

    def process_event(self, event):
        data = super().process_event(event)

        if data:
            return CustomEvent(**vars(data), toggled=self._is_toggled)
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState
from data.utils.font_helpers import text_width_to_font_size, text_height_to_font_size, height_to_font_size
from data.utils.widget_helpers import create_text_box
from data.assets import GRAPHICS

class Text(_Widget): # Pure text
    def __init__(self, text, center=True, fit_vertical=True, box_colours=None, strength=0.05, font_size=None, **kwargs):
        super().__init__(**kwargs)
        self._text = text
        self._fit_vertical = fit_vertical
        self._strength = strength
        self._box_colours = box_colours

        if fit_vertical:
            self._relative_font_size = text_height_to_font_size(self._text, self._font, (self.size[1] - 2 * (self.margin + self.border_width))) / self.surface_size[1]
        else:
            self._relative_font_size = text_width_to_font_size(self._text, self._font, (self.size[0] - 2 * (self.margin + self.border_width))) / self.surface_size[1]

        if font_size:
            self._relative_font_size = font_size / self.surface_size[1]

        self._center = center
        self.rect = self._font.get_rect(self._text, size=self.font_size)
        self.rect.topleft = self.position

        self._empty_surface = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.set_image()
        self.set_geometry()

    def resize_text(self):
        if self._fit_vertical:
            self._relative_font_size = text_height_to_font_size(self._text, self._font, (self.size[1] - 2 * (self.margin + self.border_width))) / self.surface_size[1]
        else:
            ideal_font_size = height_to_font_size(self._font, target_height=(self.size[1] - (self.margin + self.border_width))) / self.surface_size[1]
            new_font_size = text_width_to_font_size(self._text, self._font, (self.size[0] - (self.margin + self.border_width))) / self.surface_size[1]

            if new_font_size < ideal_font_size:
                self._relative_font_size = new_font_size
            else:
                self._relative_font_size = ideal_font_size

    def set_text(self, new_text):
        self._text = new_text

        self.resize_text()
        self.set_image()

    def set_image(self):
        if self._box_colours:
            self.image = create_text_box(self.size, self.border_width, self._box_colours)
        else:
            text_surface = pygame.transform.scale(self._empty_surface, self.size)
            self.image = text_surface

            if self._fill_colour:
                fill_rect = pygame.FRect(0, 0, self.size[0], self.size[1])
                pygame.draw.rect(self.image, self._fill_colour, fill_rect, border_radius=int(self.border_radius))

        self._font.strength = self._strength
        font_rect_size = self._font.get_rect(self._text, size=self.font_size).size
        if self._center:
            font_position = ((self.size[0] - font_rect_size[0]) / 2, (self.size[1] - font_rect_size[1]) / 2)
        else:
            font_position = (self.margin / 2, (self.size[1] - font_rect_size[1]) / 2)
        self._font.render_to(self.image, font_position, self._text, fgcolor=self._text_colour, size=self.font_size)

        if self._box_colours is None and self.border_width:
            fill_rect = pygame.FRect(0, 0, self.size[0], self.size[1])
            pygame.draw.rect(self.image, self._border_colour, fill_rect, width=int(self.border_width), border_radius=int(self.border_radius))

    def process_event(self, event):
        pass
```

```python
from data.widgets.bases.pressable import _Pressable
from data.widgets.bases.box import _Box
from data.widgets.text import Text
from data.constants import WidgetState, BLUE_BUTTON_COLOURS

class TextButton(_Box, _Pressable, Text):
    def __init__(self, event, **kwargs):
        _Box.__init__(self, box_colours=BLUE_BUTTON_COLOURS)
        _Pressable.__init__(
            self,
            event=event,
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=lambda: self.set_state_colour(WidgetState.BASE),
        )
        Text.__init__(self, box_colours=BLUE_BUTTON_COLOURS[WidgetState.BASE], **kwargs)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)
```

```python
import pyperclip
import pygame
from data.constants import WidgetState, CursorMode, INPUT_COLOURS
from data.components.custom_event import CustomEvent
from data.widgets.bases.pressable import _Pressable
from data.managers.logs import initialise_logger
from data.managers.animation import animation
from data.widgets.bases.box import _Box
from data.managers.cursor import cursor
from data.managers.theme import theme
from data.widgets.text import Text

logger = initialise_logger(__name__)

class TextInput(_Box, _Pressable, Text):
    def __init__(self, event, blinking_interval=530, validator=(lambda x: True), default='', placeholder='PLACEHOLDER TEXT', placeholder_colour=(200, 200, 200), cursor_colour=theme['textSecondary'], **kwargs):
        self._cursor_index = None
        # Multiple inheritance used here, adding the functionality of pressing, and custom box colours, to the text widget
        _Box.__init__(self, box_colours=INPUT_COLOURS)
        _Pressable.__init__(
            self,
            event=None,
            hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
            down_func=lambda: self.set_state_colour(WidgetState.PRESS),
            up_func=lambda: self.set_state_colour(WidgetState.BASE),
            sfx=None
        )
        Text.__init__(self, text="", center=False, box_colours=INPUT_COLOURS[WidgetState.BASE], **kwargs)

        self.initialise_new_colours(self._fill_colour)
        self.set_state_colour(WidgetState.BASE)

        pygame.key.set_repeat(500, 50)

        self._blinking_fps = 1000 / blinking_interval
        self._cursor_colour = cursor_colour
        self._cursor_colour_copy = cursor_colour
        self._placeholder_colour = placeholder_colour
        self._text_colour_copy = self._text_colour

        self._placeholder_text = placeholder
        self._is_placeholder = None
        if default:
            self._text = default
            self.is_placeholder = False
        else:
            self._text = self._placeholder_text
            self.is_placeholder = True

        self._event = event
        self._validator = validator
        self._blinking_cooldown = 0

        self._empty_cursor = pygame.Surface((0, 0), pygame.SRCALPHA)

        self.resize_text()
        self.set_image()
        self.set_geometry()

    @property
    # Encapsulated getter method
    def is_placeholder(self):
        return self._is_placeholder

    @is_placeholder.setter
    # Encapsulated setter method, used to replace text colour if placeholder text is shown
    def is_placeholder(self, is_true):
        self._is_placeholder = is_true

        if is_true:
            self._text_colour = self._placeholder_colour
        else:
            self._text_colour = self._text_colour_copy

    @property
    def cursor_size(self):
        cursor_height = (self.size[1] - self.border_width * 2) * 0.75
        return (cursor_height * 0.1, cursor_height)

    @property
    def cursor_position(self):
        current_width = (self.margin / 2)
        for index, metrics in enumerate(self._font.get_metrics(self._text, size=self.font_size)):
            if index == self._cursor_index:
                return (current_width - self.cursor_size[0], (self.size[1] - self.cursor_size[1]) / 2)

            glyph_width = metrics[4]
            current_width += glyph_width
        return (current_width - self.cursor_size[0], (self.size[1] - self.cursor_size[1]) / 2)

    @property
    def text(self):
        if self.is_placeholder:
            return ''

        return self._text

    def relative_x_to_cursor_index(self, relative_x):
        """
        Calculates cursor index using mouse position relative to the widget position.

        Args:
            relative_x (int): Horizontal distance of the mouse from the left side of the widget.

        Returns:
            int: Cursor index.
        """
        current_width = 0

        for index, metrics in enumerate(self._font.get_metrics(self._text, size=self.font_size)):
            glyph_width = metrics[4]

            if current_width >= relative_x:
                return index

            current_width += glyph_width

        return len(self._text)

    def set_cursor_index(self, mouse_pos):
        """
        Sets cursor index based on mouse position.

        Args:
            mouse_pos (list[int, int]): Mouse position relative to window screen.
        """
        if mouse_pos is None:
            self._cursor_index = mouse_pos
            return

        relative_x = mouse_pos[0] - (self.margin / 2) - self.rect.left
        relative_x = max(0, relative_x)
        self._cursor_index = self.relative_x_to_cursor_index(relative_x)

    def focus_input(self, mouse_pos):
        """
        Draws cursor and sets cursor index when user clicks on widget.

        Args:
            mouse_pos (list[int, int]): Mouse position relative to window screen.
        """
        if self.is_placeholder:
            self._text = ''
            self.is_placeholder = False

        self.set_cursor_index(mouse_pos)
        self.set_image()
        cursor.set_mode(CursorMode.IBEAM)

    def unfocus_input(self):
        """
        Removes cursor when user unselects widget.
        """
        if self._text == '':
            self._text = self._placeholder_text
            self.is_placeholder = True
            self.resize_text()

        self.set_cursor_index(None)
        self.set_image()
        cursor.set_mode(CursorMode.ARROW)

    def set_text(self, new_text):
        """
        Called by a state object to change the widget text externally.

        Args:
            new_text (str): New text to display.

        Returns:
            CustomEvent: Object containing the new text to alert state of a text update.
        """
        super().set_text(new_text)
        return CustomEvent(**vars(self._event), text=self.text)

```

```python
def process_event(self, event):
    """
    Processes Pygame events.

    Args:
        event (pygame.Event): Event to process.

    Returns:
        CustomEvent: Object containing the new text to alert state of a text update.
    """
    previous_state = self.get_widget_state()
    super().process_event(event)
    current_state = self.get_widget_state()

    match event.type:
        case pygame.MOUSEMOTION:
            if self._cursor_index is None:
                return

            # If mouse is hovering over widget, turn mouse cursor into an I-beam
            if self.rect.collidepoint(event.pos):
                if cursor.get_mode() != CursorMode.IBEAM:
                    cursor.set_mode(CursorMode.IBEAM)
            else:
                if cursor.get_mode() == CursorMode.IBEAM:
                    cursor.set_mode(CursorMode.ARROW)

            return

        case pygame.MOUSEBUTTONUP:
            # When user selects widget
            if previous_state == WidgetState.PRESS:
                self.focus_input(event.pos)
            # When user unselects widget
            if current_state == WidgetState.BASE and self._cursor_index is not None:
                self.unfocus_input()
                return CustomEvent(**vars(self._event), text=self.text)

        case pygame.KEYDOWN:
            if self._cursor_index is None:
                return

            # Handling Ctrl-C and Ctrl-V shortcuts
            if event.mod & (pygame.KMOD_CTRL):
                if event.key == pygame.K_c:
                    pyperclip.copy(self.text)
                    logger.info(f'COPIED {self.text}')

                elif event.key == pygame.K_v:
                    pasted_text = pyperclip.paste()
                    pasted_text = ''.join(char for char in pasted_text if 32 <= ord(char) <= 127)
                    self._text = self._text[:self._cursor_index] + pasted_text + self._text[self._cursor_index:]
                    self._cursor_index += len(pasted_text)

                elif event.key == pygame.K_BACKSPACE or event.key == pygame.K_DELETE:
                    self._text = ''
                    self._cursor_index = 0

                self.resize_text()
                self.set_image()
                self.set_geometry()

                return

            match event.key:
                case pygame.K_BACKSPACE:
                    if self._cursor_index > 0:
                        self._text = self._text[:self._cursor_index - 1] + self._text[self._cursor_index:]
                    self._cursor_index = max(0, self._cursor_index - 1)

                case pygame.K_RIGHT:
                    self._cursor_index = min(len(self._text), self._cursor_index + 1)

                case pygame.K_LEFT:
                    self._cursor_index = max(0, self._cursor_index - 1)

                case pygame.K_ESCAPE:
                    self.unfocus_input()
                    return CustomEvent(**vars(self._event), text=self.text)

                case pygame.K_RETURN:
                    self.unfocus_input()
                    return CustomEvent(**vars(self._event), text=self.text)

                case _:
                    if not event.unicode:
                        return

                    potential_text = self._text[:self._cursor_index] + event.unicode + self._text[self._cursor_index:]

                    # Validator lambda function used to check if inputted text is valid before displaying
                    # e.g. Time control input has a validator function checking if text represents a float
                    if self._validator(potential_text) is False:
                        return

                    self._text = potential_text
                    self._cursor_index += 1

            self._blinking_cooldown += 1
            animation.set_timer(500, lambda: self.subtract_blinking_cooldown(1))

            self.resize_text()
            self.set_image()
            self.set_geometry()

def subtract_blinking_cooldown(self, cooldown):
    """
    Subtracts blinking cooldown after certain timeframe. When blinking_cooldown is 1, cursor is able to be drawn.

    Args:
        cooldown (float): Duration before cursor can no longer be drawn.
        """
        self._blinking_cooldown = self._blinking_cooldown - cooldown

    def set_image(self):
        """
        Draws text input widget to image.
        """
        super().set_image()

        if self._cursor_index is not None:
            scaled_cursor = pygame.transform.scale(self._empty_cursor, self.cursor_size)
            scaled_cursor.fill(self._cursor_colour)
            self.image.blit(scaled_cursor, self.cursor_position)

    def update(self):
        """
        Overrides based update method, to handle cursor blinking.
        """
        super().update()
        # Calculate if cursor should be shown or not
        cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
        if cursor_frame == 1 and self._blinking_cooldown == 0:
            self._cursor_colour = (0, 0, 0, 0)
        else:
            self._cursor_colour = self._cursor_colour_copy
        self.set_image()
```

```
import pygame
from data.constants import WidgetState, Colour, BLUE_BUTTON_COLOURS, RED_BUTTON_COLOURS
from data.components.custom_event import CustomEvent
from data.managers.animation import animation
from data.widgets.text import Text

class Timer(Text):
    def __init__(self, active_colour, event=None, start_mins=60, **kwargs):
        box_colours = BLUE_BUTTON_COLOURS[WidgetState.BASE] if active_colour == Colour.BLUE else RED_BUTTON_COLOURS[WidgetState.BASE]

        self._current_ms = float(start_mins) * 60 * 1000
        self._active_colour = active_colour
        self._active = False
        self._timer_running = False
        self._event = event

        super().__init__(text=self.format_to_text(), fit_vertical=False, box_colours=box_colours, **kwargs)

    def set_active(self, is_active):
        if self._active == is_active:
            return

        if is_active and self._timer_running is False:
            self._timer_running = True
            animation.set_timer(1000, self.decrement_second)

        self._active = is_active

    def set_time(self, milliseconds):
        self._current_ms = milliseconds
        self._text = self.format_to_text()
```

```python
        self.set_image()
        self.set_geometry()

    def get_time(self):
        return self._current_ms / (1000 * 60)

    def decrement_second(self):
        if self._active:
            self.set_time(self._current_ms - 1000)

            if self._current_ms <= 0:
                self._active = False
                self._timer_running = False
                self.set_time(0)
                pygame.event.post(pygame.event.Event(pygame.MOUSEMOTION, pos=pygame.mouse.get_pos())) # RANDOM EVENT TO TRIGGER process_event
            else:
                animation.set_timer(1000, self.decrement_second)
        else:
            self._timer_running = False

    def format_to_text(self):
        raw_seconds = self._current_ms / 1000
        minutes, seconds = divmod(raw_seconds, 60)
        return f'{str(int(minutes)).zfill(2)}:{str(int(seconds)).zfill(2)}'

    def process_event(self, event):
        if self._current_ms <= 0:
            return CustomEvent(**vars(self._event), active_colour=self._active_colour)
```

```python
import pygame
from data.widgets.bases.widget import _Widget
from data.widgets.slider_thumb import _SliderThumb
from data.components.custom_event import CustomEvent
from data.constants import SettingsEventType
from data.constants import WidgetState
from data.utils.widget_helpers import create_slider
from data.utils.asset_helpers import scale_and_cache
from data.managers.theme import theme

class VolumeSlider(_Widget):
    def __init__(self, relative_length, default_volume, volume_type, thumb_colour=theme['fillSecondary'], **kwargs):
        super().__init__(relative_size=(relative_length, relative_length * 0.2), **kwargs)

        self._volume_type = volume_type
        self._selected_percent = default_volume
        self._last_mouse_x = None

        self._thumb = _SliderThumb(radius=self.size[1] / 2, border_colour=self._border_colour, fill_colour=thumb_colour)
        self._gradient_surface = create_slider(self.calculate_slider_size(), self._fill_colour, self.border_width, self._border_colour)

        self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)

    @property
    def position(self):
        '''Minus so easier to position slider by starting from the left edge of the slider instead of the thumb'''
        return (self._relative_position[0] * self.surface_size[0] - (self.size[1] / 2), self._relative_position[1] * self.surface_size[1])

    def calculate_slider_position(self):
        return (self.size[1] / 2, self.size[1] / 4)

    def calculate_slider_size(self):
        return (self.size[0] - 2 * (self.size[1] / 2), self.size[1] / 2)

    def calculate_selected_percent(self, mouse_pos):
        if self._last_mouse_x is None:
            return

        x_change = (mouse_pos[0] - self._last_mouse_x) / (self.calculate_slider_size()[0] - 2 * self.border_width)
        return max(0, min(self._selected_percent + x_change, 1))

    def calculate_thumb_position(self):
        gradient_size = self.calculate_slider_size()
        x = gradient_size[0] * self._selected_percent
        y = 0

        return (x, y)

    def relative_to_global_position(self, position):
        relative_x, relative_y = position
        return (relative_x + self.position[0], relative_y + self.position[1])

    def set_image(self):
        gradient_scaled = scale_and_cache(self._gradient_surface, self.calculate_slider_size())
        gradient_position = self.calculate_slider_position()

        self.image = pygame.transform.scale(self._empty_surface, (self.size))
        self.image.blit(gradient_scaled, gradient_position)

        thumb_position = self.calculate_thumb_position()
        self._thumb.set_surface(radius=self.size[1] / 2, border_width=self.border_width)
        self._thumb.set_position(self.relative_to_global_position((thumb_position[0], thumb_position[1])))

        thumb_surface = self._thumb.get_surface()
        self.image.blit(thumb_surface, thumb_position)

    def set_volume(self, volume):
        self._selected_percent = volume
        self.set_image()

    def process_event(self, event):
        if event.type not in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP]:
            return

        before_state = self._thumb.state
        self._thumb.process_event(event)
        after_state = self._thumb.state

        if before_state != after_state:
            self.set_image()

            if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP]:
```

```python
                self._last_mouse_x = None
                return CustomEvent(SettingsEventType.VOLUME_SLIDER_CLICK, volume=round(self._selected_percent, 3), volume_type=self._volume_type)

        if self._thumb.state == WidgetState.PRESS:
            selected_percent = self.calculate_selected_percent(event.pos)
            self._last_mouse_x = event.pos[0]

            if selected_percent:
                self._selected_percent = selected_percent
                self.set_image()
                return CustomEvent(SettingsEventType.VOLUME_SLIDER_SLIDE)
```

```python
from data.widgets.bases.widget import _Widget
from data.widgets.bases.pressable import _Pressable
from data.widgets.bases.circular import _Circular
from data.widgets.bases.box import _Box
from data.widgets.colour_display import _ColourDisplay
from data.widgets.colour_square import _ColourSquare
from data.widgets.colour_slider import _ColourSlider
from data.widgets.slider_thumb import _SliderThumb
from data.widgets.scrollbar import _Scrollbar

from data.widgets.board_thumbnail_button import BoardThumbnailButton
from data.widgets.multiple_icon_button import MultipleIconButton
from data.widgets.reactive_icon_button import ReactiveIconButton
from data.widgets.board_thumbnail import BoardThumbnail
from data.widgets.reactive_button import ReactiveButton
from data.widgets.volume_slider import VolumeSlider
from data.widgets.colour_picker import ColourPicker
from data.widgets.colour_button import ColourButton
from data.widgets.browser_strip import BrowserStrip
from data.widgets.piece_display import PieceDisplay
from data.widgets.browser_item import BrowserItem
from data.widgets.text_button import TextButton
from data.widgets.icon_button import IconButton
from data.widgets.scroll_area import ScrollArea
from data.widgets.chessboard import Chessboard
from data.widgets.text_input import TextInput
from data.widgets.rectangle import Rectangle
from data.widgets.move_list import MoveList
from data.widgets.dropdown import Dropdown
from data.widgets.carousel import Carousel
from data.widgets.switch import Switch
from data.widgets.timer import Timer
from data.widgets.text import Text
from data.widgets.icon import Icon

__all__ = ['Text', 'TextButton', 'ColourPicker', 'ColourButton', 'Switch', 'Dropdown', 'IconButton', 'Icon', 'VolumeSlider', 'TextInput', 'MultipleIconButton', 'Carousel', 'Timer', 'Rectangle', 'Chessboard', 'ScrollArea', 'MoveList', 'BoardThumbnail', 'BrowserStrip', 'BrowserItem', 'PieceDisplay', 'BoardThumbnailButton', 'ReactiveButton', 'ReactiveIconButton']
```

```python
from data.constants import WidgetState

class _Box:
    def __init__(self, box_colours):
        self._box_colours_dict = box_colours
        self._box_colours = self._box_colours_dict[WidgetState.BASE]

    def set_state_colour(self, state):
        self._box_colours = self._box_colours_dict[state]
        super().set_state_colour(state)
```

```python
from data.components.circular_linked_list import CircularLinkedList

class _Circular:
    def __init__(self, items_dict, **kwargs):
        # The key, value pairs are stored within a dictionary, while the keys to access them are stored within circular linked list.
        self._items_dict = items_dict
        self._keys_list = CircularLinkedList(list(items_dict.keys()))

    @property
    def current_key(self):
        """
        Gets the current head node of the linked list, and returns a key stored as the node data.
        Returns:
            Data of linked list head.
        """
        return self._keys_list.get_head().data

    @property
    def current_item(self):
        """
        Gets the value in self._items_dict with the key being self.current_key.

        Returns:
            Value stored with key being current head of linked list.
        """
        return self._items_dict[self.current_key]

    def set_next_item(self):
        """
        Sets the next item in as the current item.
        """
        self._keys_list.shift_head()

    def set_previous_item(self):
        """
        Sets the previous item as the current item.
        """
        self._keys_list.unshift_head()

    def set_to_key(self, key):
        """
        Sets the current item to the specified key.

        Args:
            key: The key to set as the current item.

        Raises:
            ValueError: If no nodes within the circular linked list contains the key as its data.
        """
        if self._keys_list.data_in_list(key) is False:
            raise ValueError('(_Circular.set_to_key) Key not found:', key)

        for _ in range(len(self._items_dict)):
            if self.current_key == key:
                self.set_image()
                self.set_geometry()
                return

            self.set_next_item()
```

```
import pygame
from data.constants import WidgetState
from data.managers.audio import audio
from data.assets import SFX

class _Pressable:
    def __init__(self, event, down_func=None, up_func=None, hover_func=None,
    prolonged=False, sfx=SFX['button_click'], **kwargs):
        self._down_func = down_func
        self._up_func = up_func
        self._hover_func = hover_func
        self._pressed = False
        self._prolonged = prolonged
        self._sfx = sfx

        self._event = event

        self._widget_state = WidgetState.BASE

        self._colours = {}

    def set_state_colour(self, state):
        self._fill_colour = self._colours[state]

        self.set_image()

    def initialise_new_colours(self, colour):
        r, g, b, a = pygame.Color(colour).rgba

        self._colours = {
            WidgetState.BASE: pygame.Color(r, g, b, a),
            WidgetState.HOVER: pygame.Color(min(r + 25, 255), min(g + 25, 255),
    min(b + 25, 255), a),
            WidgetState.PRESS: pygame.Color(min(r + 50, 255), min(g + 50, 255),
    min(b + 50, 255), a)
        }

    def get_widget_state(self):
        return self._widget_state

    def process_event(self, event):
        match event.type:
            case pygame.MOUSEBUTTONDOWN:
                if self.rect.collidepoint(event.pos):
                    self._down_func()
                    self._widget_state = WidgetState.PRESS

            case pygame.MOUSEBUTTONUP:
                if self.rect.collidepoint(event.pos):
                    if self._widget_state == WidgetState.PRESS:
                        if self._sfx:
                            audio.play_sfx(self._sfx)

                        self._up_func()
                        self._widget_state = WidgetState.HOVER
                        return self._event

                    elif self._widget_state == WidgetState.BASE:
```

```python
                        self._hover_func()

                elif self._prolonged and self._widget_state == WidgetState.PRESS:
                    if self._sfx:
                        audio.play_sfx(self._sfx)
                    self._up_func()
                    self._widget_state = WidgetState.BASE
                    return self._event

            case pygame.MOUSEMOTION:
                if self.rect.collidepoint(event.pos):
                    if self._widget_state == WidgetState.PRESS:
                        return
                    elif self._widget_state == WidgetState.BASE:
                        self._hover_func()
                        self._widget_state = WidgetState.HOVER
                    elif self._widget_state == WidgetState.HOVER:
                        self._hover_func()
                else:
                    if self._prolonged is False:
                        if self._widget_state in [WidgetState.PRESS, WidgetState.HOVER]:
                            self._widget_state = WidgetState.BASE
                            self._up_func()
                        elif self._widget_state == WidgetState.BASE:
                            return
                    elif self._prolonged is True:
                        if self._widget_state in [WidgetState.PRESS, WidgetState.BASE]:
                            return
                        else:
                            self._widget_state = WidgetState.BASE
                            self._up_func()
```

```python
import pygame
from data.constants import SCREEN_SIZE
from data.managers.theme import theme
from data.assets import DEFAULT_FONT

DEFAULT_SURFACE_SIZE = SCREEN_SIZE
REQUIRED_KWARGS = ['relative_position', 'relative_size']

class _Widget(pygame.sprite.Sprite):
    def __init__(self, **kwargs):
        """
        Every widget has the following attributes:

        surface (pygame.Surface): The surface the widget is drawn on.
        raw_surface_size (tuple[int, int]): The initial size of the window screen, remains constant.
        parent (_Widget, optional): The parent widget position and size is relative to.

        Relative to current surface:
        relative_position (tuple[float, float]): The position of the widget relative to its surface.
        relative_size (tuple[float, float]): The scale of the widget relative to its surface.

        Remains constant, relative to initial screen size:
        relative_font_size (float, optional): The relative font size of the widget.
    relative_margin (float): The relative margin of the widget.
    relative_border_width (float): The relative border width of the widget.
    relative_border_radius (float): The relative border radius of the widget.

    anchor_x (str): The horizontal anchor direction ('left', 'right', 'center').
    anchor_y (str): The vertical anchor direction ('top', 'bottom', 'center').
    fixed_position (tuple[int, int], optional): The fixed position of the widget in pixels.
    border_colour (pygame.Color): The border color of the widget.
    text_colour (pygame.Color): The text color of the widget.
    fill_colour (pygame.Color): The fill color of the widget.
    font (pygame.freetype.Font): The font used for the widget.
    """
    super().__init__()

    for required_kwarg in REQUIRED_KWARGS:
        if required_kwarg not in kwargs:
            raise KeyError(f'(_Widget.__init__) Required keyword "{required_kwarg}" not in base kwargs')

    self._surface = None # Set in WidgetGroup, as needs to be reassigned every frame
    self._raw_surface_size = DEFAULT_SURFACE_SIZE

    self._parent = kwargs.get('parent')

    self._relative_font_size = None # Set in subclass

    self._relative_position = kwargs.get('relative_position')
    self._relative_margin = theme['margin'] / self._raw_surface_size[1]
    self._relative_border_width = theme['borderWidth'] / self._raw_surface_size[1]
    self._relative_border_radius = theme['borderRadius'] / self._raw_surface_size[1]

    self._border_colour = pygame.Color(theme['borderPrimary'])
    self._text_colour = pygame.Color(theme['textPrimary'])
    self._fill_colour = pygame.Color(theme['fillPrimary'])
    self._font = DEFAULT_FONT

    self._anchor_x = kwargs.get('anchor_x') or 'left'
    self._anchor_y = kwargs.get('anchor_y') or 'top'
    self._fixed_position = kwargs.get('fixed_position')
    scale_mode = kwargs.get('scale_mode') or 'both'

    if kwargs.get('relative_size'):
        match scale_mode:
            case 'height':
                self._relative_size = kwargs.get('relative_size')
            case 'width':
                self._relative_size = ((kwargs.get('relative_size')[0] * self.surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] * self.surface_size[0]) / self.surface_size[1])
            case 'both':
                self._relative_size = ((kwargs.get('relative_size')[0] * self.surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
            case _:
                raise ValueError('(_Widget.__init__) Unknown scale mode:', scale_mode)
    else:
        self._relative_size = (1, 1)
```

```python

        if 'margin' in kwargs:
            self._relative_margin = kwargs.get('margin') / self._raw_surface_size[1]

            if (self._relative_margin * 2) > min(self._relative_size[0], self._relative_size[1]):
                raise ValueError('(_Widget.__init__) Margin larger than specified size!')

        if 'border_width' in kwargs:
            self._relative_border_width = kwargs.get('border_width') / self._raw_surface_size[1]

        if 'border_radius' in kwargs:
            self._relative_border_radius = kwargs.get('border_radius') / self._raw_surface_size[1]

        if 'border_colour' in kwargs:
            self._border_colour = pygame.Color(kwargs.get('border_colour'))

        if 'fill_colour' in kwargs:
            self._fill_colour = pygame.Color(kwargs.get('fill_colour'))

        if 'text_colour' in kwargs:
            self._text_colour = pygame.Color(kwargs.get('text_colour'))

        if 'font' in kwargs:
            self._font = kwargs.get('font')

    @property
    def surface_size(self):
        """
        Gets the size of the surface widget is drawn on.
        Can be either the window size, or another widget size if assigned to a parent.

        Returns:
            tuple[int, int]: The size of the surface.
        """
        if self._parent:
            return self._parent.size
        else:
            return self._raw_surface_size

    @property
    def position(self):
        """
        Gets the position of the widget.
        Accounts for fixed position attribute, where widget is positioned in pixels regardless of screen size.
        Acounts for anchor direction, where position attribute is calculated relative to one side of the screen.

        Returns:
            tuple[int, int]: The position of the widget.
        """
        x, y = None, None
        if self._fixed_position:
            x, y = self._fixed_position
        if x is None:
            x = self._relative_position[0] * self.surface_size[0]
        if y is None:
            y = self._relative_position[1] * self.surface_size[1]

        if self._anchor_x == 'left':
            x = x
        elif self._anchor_x == 'right':
            x = self.surface_size[0] - x - self.size[0]
        elif self._anchor_x == 'center':
            x = (self.surface_size[0] / 2 - self.size[0] / 2) + x

        if self._anchor_y == 'top':
            y = y
        elif self._anchor_y == 'bottom':
            y = self.surface_size[1] - y - self.size[1]
        elif self._anchor_y == 'center':
            y = (self.surface_size[1] / 2 - self.size[1] / 2) + y

        # Position widget relative to parent, if exists.
        if self._parent:
            return (x + self._parent.position[0], y + self._parent.position[1])
        else:
            return (x, y)

    @property
    def size(self):
        return (self._relative_size[0] * self.surface_size[1], self._relative_size[1] * self.surface_size[1])

    @property
    def margin(self):
        return self._relative_margin * self._raw_surface_size[1]

    @property
    def border_width(self):
        return self._relative_border_width * self._raw_surface_size[1]

    @property
    def border_radius(self):
        return self._relative_border_radius * self._raw_surface_size[1]

    @property
    def font_size(self):
        return self._relative_font_size * self.surface_size[1]

    def set_image(self):
        """
        Abstract method to draw widget.
        """
        raise NotImplementedError

    def set_geometry(self):
        """
        Sets the position and size of the widget.
        """
        self.rect = self.image.get_rect()

        if self._anchor_x == 'left':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'right':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'center':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position

    def set_surface_size(self, new_surface_size):
        """
        Sets the new size of the surface widget is drawn on.

        Args:
            new_surface_size (tuple[int, int]): The new size of the surface.
        """
        self._raw_surface_size = new_surface_size

    def process_event(self, event):
        """
        Abstract method to handle events.

        Args:
            event (pygame.Event): The event to process.
        """
        raise NotImplementedError
```