Chapter 1

Technical Solution

1.1 File Tree Diagram

To help navigate through the source code, I have included the following directory tree diagram, along with comments to explain the general purpose of code contained within specifc directories and Python files.

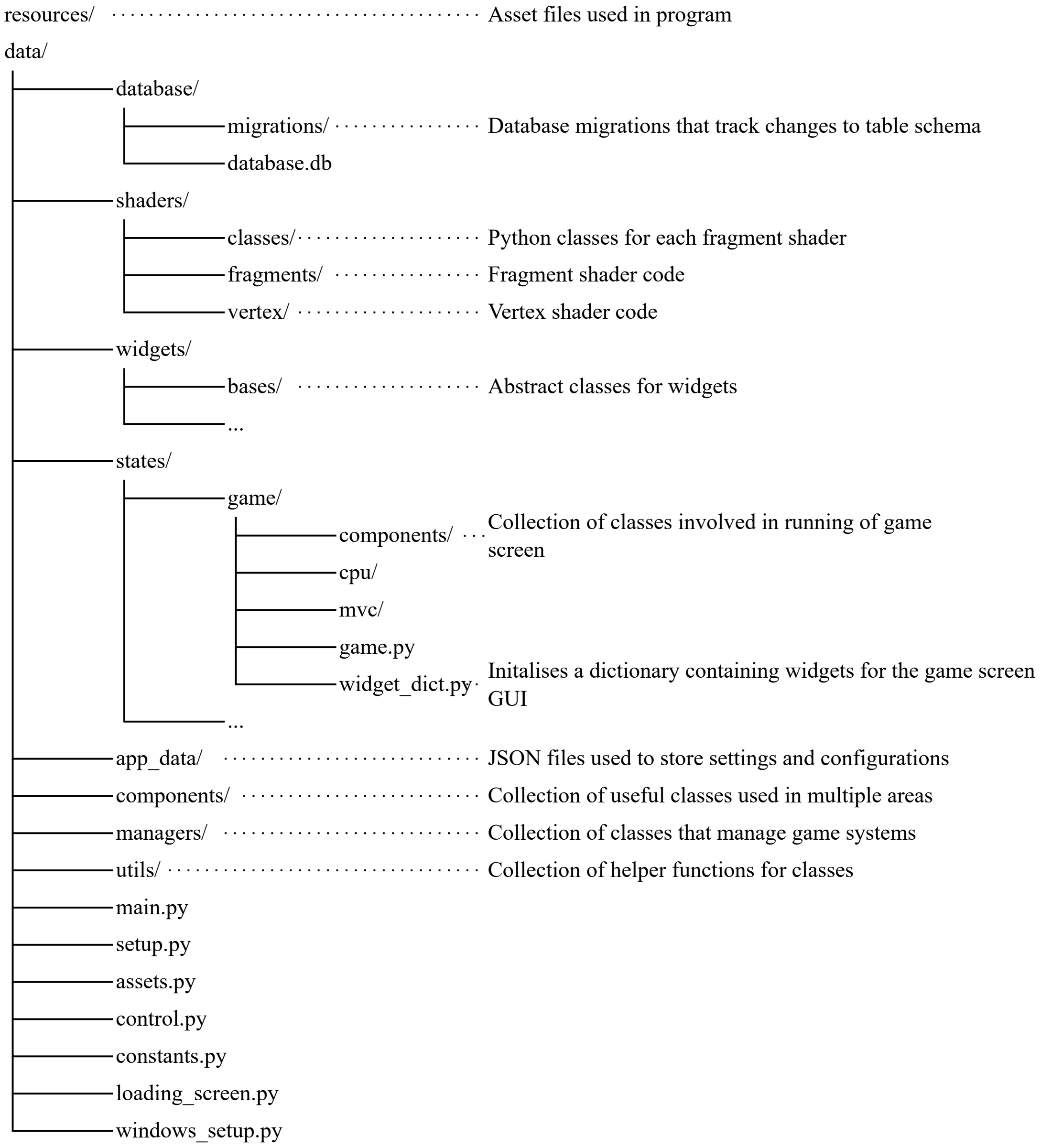
resources/ Asset files used in program
data/
database/
migrations/ Database migrations that track changes to table schema
database.db
shaders/
classes/ Python classes for each fragment shader
fragments/ Fragment shader code
vertex/ Vertex shader code
widgets/
bases/ Abstract classes for widgets
...
states/
game/
components/ Collection of classes involved in running of game screen
cpu/
mvc/
game.py
widget_dict.py Initalises a dictionary containing widgets for the game screen GUI
...
app_data/ JSON files used to store settings and configurations
components/ Collection of useful classes used in multiple areas
managers/ Collection of classes that manage game systems
utils/ Collection of helper functions for classes
main.py
setup.py
assets.py
control.py
constants.py
loading_screen.py
windows_setup.py

Figure 1.1: File tree diagram

1.2 Summary of Complexity

- Minimax improvements (1.6.2 and 1.6.3 and 1.6.4)
- Shadow mapping and coordinate transformations (1.9.3)
- Recursive Depth-First Search tree traversal (1.3.4 and 1.6.1)
- Circular doubly-linked list and stack (1.4.3 and 1.7.1)
- Multipass shaders and gaussian blur (1.9.2)
- Aggregate and Window SQL functions (1.8.2)
- OOP techniques (1.4.3 and 1.4.4)
- Multithreading (1.3.2 and 1.6.6)
- Bitboards (1.5.5)
- Zobrist hashing (1.6.7)
- (File handling and JSON parsing) (1.3.3)
- (Dictionary recursion) (1.3.4)
- (Dot product) (1.3.3 and 1.9.2)

1.3 Overview

1.3.1 Main

The file `main.py` is run by the root file `run.py`. Here resources-intensive classes such as the state and asset files are initialised, while the program displays a loading screen to hide the loading process. The main game loop is then executed.

`main.py`

```python
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()
    }

    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()
```

1.3.2 Loading Screen

Multithreading is used to separate the loading screen GUI from the resources intensive actions in `main.py`, to keep the GUI responsive. The easing function `easeOutBack` is also used to animate the logo.

loading_screen.py

```
import pygame
import threading
import sys
from pathlib import Path
from data.helpers.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):
    """
```

```python
    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)
```

1.3.3 Helper functions

These files provide useful functions for different classes.

`asset_helpers.py (Functions used for assets and pygame Surfaces)`

```
import pygame
from PIL import Image
from functools import cache
from random import 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)
```

```python

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:
        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.
    """
    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
```

data_helpers.py (Functions used for file handling and JSON parsing)

```
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)
```

```python

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)')
```

widget_helpers.py (Files used for creating widgets)

```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

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
```

1.3.4 Theme

The theme manager file is responsible for providing an instance where the colour palette and dimensions for the GUI can be accessed. Values read from a JSON file are **recursively** flattened, with keys created from the dictionary hierarchy, and stored into the internal dictionary of a `ThemeManager` object.

`theme.py`

```
from data.helpers.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()
```

1.4 GUI

1.4.1 Laser

The `LaserDraw` class draws the laser in both the game and review screens.

laser_draw.py

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

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
            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
```

1.4.2 Particles

The `ParticlesDraw` class draws particles in both the game and review screens. The particles are either fragmented pieces when destroyed, or laser particles emitted from the Sphinx. Particles are given custom velocity, rotation, opacity and size parameters.

`particles_draw.py`

```
import pygame
from random import randint
from data.helpers.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):
```

```python
        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]
            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
        ])
```

1.4.3 Widget Bases

Widget bases are used as the base classes for for my widgets system. They contain both attributes and getter methods that provide both basic functionalities such as size and position, and abstract methods to be overriden. These bases are designed to be used with **multiple inheritance**, where multiple bases can be combined to add functionality to the final widget. **Encapsulation** also allows me to simplify interactions between widgets, as using getter methods instead of protected attributes allows me to add logic while accessing an attribute, such as in `widget.py`, where the logic to fetch the parent surface instead of the windows screen is hidden within the base class.

Widget

All widgets are a subclass of the `Widget` class.
`widget.py`

```
import pygame
from data.utils.constants import SCREEN_SIZE
from data.managers.theme import theme
from data.utils.assets import DEFAULT_FONT

DEFAULT_SURFACE_SIZE = SCREEN_SIZE
REQUIRED_KWARGS = ['relative_position', 'relative_size']
```

```python

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)

        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])

        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
```

```python
    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
```

Circular

The `Circular` class provides an internal circular linked list, giving functionality to support widgets which rotate between text/icons. `circular.py`

```python
from data.components.circular_linked_list import CircularLinkedList
```

```python

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()
```

Circular Linked List

As described in Section ??, the `CircuarLinkedList` class implements a **circular doubly-linked list**. Used for the internal logic of the `Circular` class.

circular_linked_list.py

```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.

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

1.4.4 Widgets

As described in Section ??, each state contains a `WIDGET_DICT` map, which contains and initialises each widget with their own attributes, and provides references to run methods on them in the state code. Each `WIDGET_DICT` is passed into a `WidgetGroup` object, which is responsible for drawing, resizing and handling all widgets for the current state. Below is a list of all the widgets I have implemented (See Section ??):

- BoardThumbnailButton
- MultipleIconButton
- ReactiveIconButton
- BoardThumbnail
- ReactiveButton
- VolumeSlider
- ColourPicker
- ColourButton
- BrowserStrip
- PieceDisplay
- BrowserItem
- TextButton
- IconButton
- ScrollArea
- Chessboard
- TextInput
- Rectangle
- MoveList
- Dropdown
- Carousel
- Switch
- Timer
- Text
- Icon
- (_ColourDisplay)
- (_ColourSquare)
- (_ColourSlider)
- (_SliderThumb)
- (_Scrollbar)

CustomEvent

The `CustomEvent` class is used to pass data between states and widgets. An event argument is passed into interactive widgets; When a widget wants to pass data back to the state, it returns the event, and adds any attributes that is required. The state then receives and handles these returned events accordingly.

`custom_event.py`

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

# Required keyword arguments when creating a CustomEvent object with a specific EventType
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)
```

ReactiveIconButton

The `ReactiveIconButton` widget is a pressable button that changes the icon displayed when it is hovered or pressed.

`reactive_icon_button.py`

```python
from data.widgets.reactive_button import ReactiveButton
from data.utils.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
        )
```

ReactiveButton

The `ReactiveButton` widget is the parent class for `ReactiveIconButton`. It provides the methods for clicking, rotating between widget states, positioning etc.

`reactive_button.py`

```python
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.utils.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.
        """
        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)
```

ColourSlider

The `ColourSlider` widget is instanced in the `ColourPicker` class. It provides a slider for changing between hues for the colour picker, using the functionality of the `SliderThumb` class.

`colour_slider.py`

```python
import pygame
from data.helpers.widget_helpers import create_slider_gradient
from data.helpers.asset_helpers import smoothscale_and_cache
from data.widgets.slider_thumb import _SliderThumb
from data.widgets.bases.widget import _Widget
from data.utils.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 initialised 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
```

TextInput

The `TextInput` widget is used for inputting fen strings and time controls.

text_input.py

```python
import pyperclip
import pygame
from data.utils.constants import WidgetState, 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.utils.enums import CursorMode
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.
```

```python
        """
        super().set_text(new_text)
        return CustomEvent(**vars(self._event), text=self.text)

    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()
```

1.5 Game

1.5.1 Model

As described in Section **??**, this is the model class for my implementation of a **MVC architecture** for the game screen. It is responsible for processing user inputs through the game controller, processing the board and CPU, and sending information through the view class.

game_model.py

```python
from random import getrandbits
from data.states.game.components.fen_parser import encode_fen_string
from data.states.game.widget_dict import GAME_WIDGETS
from data.states.game.cpu.cpu_thread import CPUThread
from data.components.custom_event import CustomEvent
from data.helpers.bitboard_helpers import is_occupied
from data.helpers import input_helpers as ip_helpers
from data.states.game.components.board import Board
from data.states.game.components.move import Move
from data.utils.event_types import GameEventType
from data.managers.logs import initialise_logger
from data.managers.animation import animation
from data.states.game.cpu.engines import *
from data.utils.constants import EMPTY_BB
from data.utils.enums import Colour

logger = initialise_logger(__name__)

```

```python
CPU_LIMIT_MS = 15000

class GameModel:
    def __init__(self, game_config):
        self._listeners = {
            'game': [],
            'win': [],
            'pause': [],
        }
        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._board = Board(fen_string=game_config['FEN_STRING'])

        self._cpu = IDMinimaxCPU(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']
        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

    # Employ time management system to kill search if using an iterative
deepening CPU
    if isinstance(self._cpu, IDMinimaxCPU):
        move_id = getrandbits(32)
        self._cpu_thread.start_cpu(self.get_board(), id=move_id)
        animation.set_timer(CPU_LIMIT_MS, lambda: self._cpu_thread.stop_cpu(id
=move_id))
    else:
        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 thread by 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
```

1.5.2 View

As described in Section ??, the view class is responsible for displaying changes to information regarding the gameplay. The `process_model_event` procedure is registered with the model class, which executes it whenever the display needs to be updated (e.g. piece move), and the appropiate handling function within the view class is called by mapping the event type to the corresponding handler function.

game_view.py

```python
import pygame
from data.utils.enums import 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.helpers.bitboard_helpers import bitboard_to_coords
from data.helpers.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.utils.event_types import GameEventType
from data.managers.window import window
from data.managers.audio import audio
from data.utils.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)
        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("CPU calculating a crazy move...")
            case StatusText.WIN:
                if self._model.states['WINNER'] == Miscellaneous.DRAW:
                    GAME_WIDGETS['status_text'].set_text("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("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):
        """
        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 appropriate 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

```

```python
    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):
        """
        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)
```

1.5.3 Controller

As described in Section ??, the controller class is responsible for receiving external input through Pygame events, and processing them via the model and view classes.

game_controller.py

```
import pygame
from data.helpers import bitboard_helpers as bb_helpers
from data.utils.enums import MoveType, Miscellaneous
from data.states.game.components.move import Move
from data.utils.event_types import GameEventType
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

            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)

```

```python
        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
```

1.5.4 Board

The `Board` class implements the Laser Chess board, and is responsible for handling moves, captures, and win conditions.

board.py

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


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:
```

```python
                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(f'Invalid move - no piece found on source square. {move}')
        elif piece_symbol == Piece.SPHINX:
            raise ValueError(f'Invalid move - sphinx piece is immovable. {move}')

        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
```

```python

        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):
        """
        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)
```

1.5.5 Bitboards

As described in Section ??, the `BitboardCollection` class uses helper functions found in `bitboard_helpers.py` such as `pop_count`, to initialise and manage bitboard transformations.
bitboard_collection.py

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

logger = initialise_logger(__name__)

```

```python
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.
```

```python
            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)
        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.

        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
```

1.6 CPU

This section includes my implementation for the CPU engine run on minimax, including its various improvements and accessory classes.

Every CPU engine class is a subclass of a `BaseCPU` abstract class, and therefore contains the same attribute and method names. This means **polymorphism** can be used again to easily to test and vary the difficulty by switching out which CPU engine is used.

The method `find_move` is called by the CPU thread. `search` is then called recursively to traverse the minimax tree, and find an optimal move. The move is then return to `find_move` and passed and run with the callback function. A `stats` dictionary is also created in the base class, used to collect information for each search.

1.6.1 Minimax

As described in Section **??**, the minimax engine uses **DFS** to traverse the game tree and evaluate node accordingly, by **recursively** calling the `search` function.

`minimax.py`

```
from random import choice
from data.states.game.cpu.base import BaseCPU
from data.utils.enums import Score, Colour

```

```python
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

            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
```

1.6.2 Alpha-beta Pruning

As described in Section ??, the `ABMinimaxCPU` class introduces pruning to reduce the number of nodes evaluated during a minimax search.

`alpha_beta.py`

```
from data.states.game.cpu.move_orderer import MoveOrderer
from data.states.game.cpu.base import BaseCPU
from data.utils.enums import Score, Colour

class ABMinimaxCPU(BaseCPU):
    def __init__(self, max_depth, callback, verbose=True):
        super().__init__(callback, verbose)
        self._max_depth = max_depth
        self._orderer = MoveOrderer()

    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, laser_coords=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 self._orderer.get_moves(board, hint=hint, laser_coords=laser_coords):
                laser_result = board.apply_move(move)
                new_score = self.search(board, depth - 1, alpha, beta, stop_event, laser_coords=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

```

```python
            for move in self._orderer.get_moves(board, hint=hint, laser_coords=laser_coords):
                laser_result = board.apply_move(move)
                new_score = self.search(board, depth - 1, alpha, beta, stop_event, laser_coords=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
```

1.6.3 Transposition Table

For adding transposition table functionality to my other engine classes, as described in Section ??, I have decided to use a mixin design architecture. This allows me to **reuse code** by adding mixins to many different classes, and inject additional transposition table methods and functionality into other engines.

`transposition_table.py`

```python
from data.states.game.cpu.transposition_table import TranspositionTable
from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU

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, laser_coords=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)
```

1.6.4 Iterative Deepening

As described in ??, the depth for each search is increased for each iteration through the for loop, with the best move found on one depth being used as the starting move for the following depth.

iterative_deepening.py

```
from copy import deepcopy
from random import choice
from data.states.game.cpu.engines.transposition_table import TranspositionTableMixin
from data.states.game.cpu.transposition_table import TranspositionTable
from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU
from data.managers.logs import initialise_logger
from data.utils.enums import Score

logger = initialise_logger(__name__)

class IterativeDeepeningMixin:
    def find_move(self, board, stop_event):
        """
        Iterates through increasing depths to find the best move.

        Args:
            board (Board): The current board state.
            stop_event (threading.Event): Event used to kill search from an external class.
        """
        self._table = TranspositionTable()

        best_move = None

        for depth in range(1, self._max_depth + 1):
            self.initialise_stats()

            # Use copy of board as search can be terminated before all tested moves are undone
            board_copy = deepcopy(board)

            try:
                best_score, best_move = self.search(board_copy, depth, -Score.INFINITE, Score.INFINITE, stop_event, hint=best_move)
            except TimeoutError:
                # If allocated time is up, use previous depth's best move
                logger.info(f'Terminated CPU search early at depth {depth}. Using existing best move: {best_move}')

                if best_move is None:
                    # If search is terminated at depth 0, use random move
                    best_move = choice(board_copy.generate_all_moves())
                    logger.warning('CPU terminated before any best move found! Using random move.')

                break

            self._stats['ID_depth'] = depth

        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)
```

1.6.5 Evaluator

As described in Section ??, I have opted to separate the evaluation class into separate methods for each aspect of the evaluation, and amalgamating all of them to form one unified `evaluate` function, as this allows me to debug each function easily.

evaluator.py

```
from data.helpers.bitboard_helpers import pop_count, occupied_squares, bitboard_to_index
from data.states.game.components.psqt import PSQT, FLIP
from data.utils.enums import Colour, Piece, Score
```

```python
from data.managers.logs import initialise_logger

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
```

1.6.6 Multithreading

As described in Section ??, when the game starts, a `CPUThread` object is created with the selected CPU. The `start` method is called whenever it is the CPU's turn, passing the board as an argument to work on. Each run is also given a random ID, to ensure that only the right search is able to be forcibly terminated early. Using **multithreading** allows the game MVC to continue running smoothly while the CPU calculates its moves on a separate thread.

`cpu_thread.py`

```
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
        self._id = None

    def kill_thread(self):
        """
        Kills the CPU and terminates the thread by stopping the run loop.
        """
        self.stop_cpu(force=True)
        self._running = False

    def stop_cpu(self, id=None, force=False):
        """
        Kills the CPU's move search.

        Args:
            id (int, optional): Id of search to kill, only kills if matching.
            force (bool, optional): Forcibly kill search regardless of id.
        """
        if self._id == id or force:
            self._stop_event.set()
            self._board = None

    def start_cpu(self, board, id=None):
        """
        Starts the CPU's move search.

        Args:
            board (Board): The current board state.
            id (int, optional): Id of current search.
        """
        self._stop_event.clear()
        self._board = board
        self._id = id

    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...')
```

1.6.7 Zobrist Hashing

As described in Section ??, the `ZobristHasher` class provides methods to successivly **hash** a given board for every move played, with the initial hash being generated in the `Board` class.

zobrist_hasher.py

```
from random import randint
from data.helpers.bitboard_helpers import bitboard_to_index
from data.utils.enums 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)]
# 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.
```

```python

        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 the rotation on 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
```

1.6.8 Cache

As described in Section ??, the `TranspositionTable` class maintains an internal hash map to store already evaluated board positions. Since I have chosen to use a dictionary instead of an array, the Zobrist hash for the board can be used as the keys for the dictionary as is, as it doesn't correspond to the index position as will be the case if I use an array.
`transposition_table.py`

```python
from data.utils.enums 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

    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.
        """
        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
```

1.7 States

To switch between different screens, I have decided to use a state machine design pattern. This ensures that there is only one main game loop controlling movement between states, handled with the `Control` object. All `State` object contain a `next` and `previous` attribute to tell the `Control` class which screen to switch to, which also calls all `State` methods accordingly.

The `startup` method is called when switched to a new state, and `cleanup` when exiting. Within the `startup` function, the state widgets dictionary is passed into a `WidgetGroup` object. The `process_event` method is called on the `WidgetGroup` every frame to process user input, and handle the returned events accordingly. The `WidgetGroup` object can therefore be thought of as a controller, and the state as the model, and the widgets as the view.

1.7.1 Review

The `Review` state uses this logic to allow users to scroll through moves in their past games. All moves are stored in two stacks, as described in Section ??, and exchanged using `pop` and `append` (push) methods.

review.py

```
import pygame
from collections import deque
```

```python
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.helpers.bitboard_helpers import bitboard_to_coords
from data.helpers.browser_helpers import get_winner_string
from data.states.review.widget_dict import REVIEW_WIDGETS
from data.states.game.components.board import Board
from data.utils.event_types import ReviewEventType
from data.components.game_entry import GameEntry
from data.managers.logs import initialise_logger
from data.utils.constants import ShaderType
from data.managers.window import window
from data.utils.assets import MUSIC
from data.utils.enums import Colour
from data.control import _State

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):
        """
        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)
```

```python

        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:
                    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)
```

1.8 Database

This section outlines my database implementation using the Python module sqlite3.

1.8.1 DDL

As mentioned in Section ??, the `migrations` directory contains a collection of Python scripts that edit the game table schema. The files are named with a description of their changes and datetime for organisational purposes.
`create_games_table_19112024.py`

```
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()
```

Using the `ALTER` command allows me to rename table columns.

change_fen_string_column_name_23122024.py

```python
import sqlite3
from pathlib import Path

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

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

1.8.2 DML

As described in Section ??, this file provides functions to help modify the database, with **Aggregate** and **Window** commands used to retrieve the number of rows and sort them to be returned. database_helpers.py

```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.
```

```python

    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()
```

1.9 Shaders

1.9.1 Shader Manager

The `ShaderManager` class is responsible for handling all shader passes, handling the Pygame display, and combining both and drawing the result to the window screen. The class also **inherits** from the `SMProtocol` class, an **interface** class containing all required `ShaderManager` methods and attributes to aid with syntax highlighting in the fragment shader classes.

Fragment shaders such as `Bloom` are applied by default, and others such as `Ray` are applied during runtime through calling methods on `ShaderManager`, and adding the appropiate fragment shader class to the internal shader pass list.

`shader.py`

```
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.utils.constants import ShaderType

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.
```

```python

        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 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)
```

1.9.2 Bloom

The `Bloom` shader effect is a common shader effect giving the illusion of a bright light. It consists of blurred fringes of light extending from the borders of bright areas. This effect can be achieved through obtaining all bright areas of the image, applying a Gaussian blur, and blending the blur additively onto the original image.

My `ShaderManager` class works with this multi-pass shader approach by reading the texture from the last shader's framebuffer for each pass.

Extracting bright colours

The `highlight_brightness` fragment shader extracts all colours that are bright enough to exert the bloom effect.

highlight_brightness.frag

```
# version 330 core

in vec2 uvs;
out vec4 f_colour;

uniform sampler2D image;
uniform float threshold;
uniform float intensity;

void main() {
    vec4 pixel = texture(image, uvs);
    // Dot product used to calculate brightness of a pixel from its RGB values
    // Values taken from https://en.wikipedia.org/wiki/Relative_luminance
    float brightness = dot(pixel.rgb, vec3(0.2126, 0.7152, 0.0722));
    float isBright = step(threshold, brightness);

    f_colour = vec4(vec3(pixel.rgb * intensity) * isBright, 1.0);
}
```

Blur

The `Blur` class implements a two-pass **Gaussian blur**. This is preferably over a one-pass blur, as the complexity is $O(2n)$, sampling n pixels twice, as opposed to $O(n^2)$. I have implemented this using the ping-pong technique, with the first pass for blurring the image horizontally, and the second pass for blurring vertically, and the resultant textures being passed repeatedly between two framebuffers.

blur.py

```python
from data.shaders.protocol import SMProtocol
from data.utils.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"))
```

blur.frag

```glsl
// Modified from https://learnopengl.com/Advanced-Lighting/Bloom
#version 330 core

in vec2 uvs;
```

```
out vec4 f_colour;

uniform sampler2D image;
uniform bool horizontal;
uniform int passes;
uniform float weight[5] = float[] (0.227027, 0.1945946, 0.1216216, 0.054054, 0.016216);

void main() {
    vec2 offset = 1.0 / textureSize(image, 0);
    vec3 result = texture(image, uvs).rgb * weight[0];

    if (horizontal) {
        for (int i = 1; i < passes; ++i) {
            result += texture(image, uvs + vec2(offset.x * i, 0.0)).rgb * weight[i];
            result += texture(image, uvs - vec2(offset.x * i, 0.0)).rgb * weight[i];
        }
    }
    else {
        for (int i = 1; i < passes; ++i) {
            result += texture(image, uvs + vec2(0.0, offset.y * i)).rgb * weight[i];
            result += texture(image, uvs - vec2(0.0, offset.y * i)).rgb * weight[i];
        }
    }

    f_colour = vec4(result, 1.0);
}
```

Combining

The `Bloom` class combines the two operations, taking the highlighted areas, blurs them, and adds the RGB values for the final result onto the original texture to simulate bloom.
`bloom.py`

```
from data.shaders.classes.highlight_brightness import _HighlightBrightness
from data.shaders.classes.highlight_colour import _HighlightColour
from data.shaders.protocol import SMProtocol
from data.shaders.classes.blur import _Blur
from data.utils.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)

```

```python
    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)
```

1.9.3 Rays

As described in Section ??, the `Ray` shader is applied whenever the sphinx shoots a laser. It simulates a 2D light source, providing pixel perfect shadows, through the shadow mapping technique outlined in Section ??. The laser demo seen on the main menu screen is also achieved using the Ray shader, by clamping the angle at which it emits light to a narrower range.

Occlusion

The occlusion fragment shader processes all pixels with a given colour value as being occluding.
`occlusion.frag`

```
# version 330 core

in vec2 uvs;
out vec4 f_colour;

uniform sampler2D image;
uniform vec3 checkColour;

void main() {
    vec4 pixel = texture(image, uvs);

    // If pixel is occluding colour, set pixel to white
    if (pixel.rgb == checkColour) {
        f_colour = vec4(1.0, 1.0, 1.0, 1.0);
    // Else, set pixel to black
    } else {
        f_colour = vec4(vec3(0.0), 1.0);
    }
}
```

Shadowmap

The shadowmap fragment shader takes the occluding texture and creates a 1D shadow map.
`shadowmap.frag`

```
# version 330 core

#define PI 3.1415926536;

in vec2 uvs;
out vec4 f_colour;

uniform sampler2D image;
uniform float resolution;
uniform float THRESHOLD=0.99;

void main() {
  float maxDistance = 1.0;

    for (float y = 0.0 ; y < resolution ; y += 1.0) {
        //rectangular to polar filter
        float currDistance = y / resolution;

        vec2 norm = vec2(uvs.x, currDistance) * 2.0 - 1.0; // Range from [0, 1] -> [-1, 1]
        float angle = (1.5 - norm.x) * PI; // Range from [-1, 1] -> [0.5PI, 2.5PI]
        float radius = (1.0 + norm.y) * 0.5; // Range from [-1, 1] -> [0, 1]

        //coord which we will sample from occlude map
        vec2 coords = vec2(radius * -sin(angle), radius * -cos(angle)) / 2.0 + 0.5;

        // Sample occlusion map
        vec4 occluding = texture(image, coords);

        // If pixel is not occluding (Red channel value below threshold), set maxDistance to current distance
    // If pixel is occluding, don't change distance
    // maxDistance therefore is the distance from the center to the nearest occluding pixel
        maxDistance = max(maxDistance * step(occluding.r, THRESHOLD), min(maxDistance, currDistance));
    }

    f_colour = vec4(vec3(maxDistance), 1.0);
}
```

Lightmap

The lightmap shader checks if a pixel is in shadow, blurs the result, and applies the radial light source.

lightmap.frag

```
# version 330 core

#define PI 3.14159265

in vec2 uvs;
out vec4 f_colour;

uniform float softShadow;
uniform float resolution;
uniform float falloff;
uniform vec3 lightColour;
uniform vec2 angleClamp;
uniform sampler2D occlusionMap;
uniform sampler2D image;

vec3 normLightColour = lightColour / 255;
vec2 radiansClamp = angleClamp * (PI / 180);

float sample(vec2 coord, float r) {
  /*
  Sample from the 1D distance map.

  Returns:
    float: 1.0 if sampled radius is greater than the passed radius, 0.0 if not.
  */
  return step(r, texture(image, coord).r);
}

void main() {
  // Cartesian to polar transformation
  // Range from [0, 1] -> [-1, 1]
  vec2 norm = uvs.xy * 2.0 - 1.0;
  float angle = atan(norm.y, norm.x);
  float r = length(norm);

  // The texture coordinates to sample our 1D lookup texture
  // Always 0.0 on y-axis, as the texture is 1D
  float x = (angle + PI) / (2.0 * PI); // Normalise angle to [0, 1]
  vec2 tc = vec2(x, 0.0);

  // Sample the 1D lookup texture to check if pixel is in light or in shadow
  // Gives us hard shadows
  // 1.0 -> in light, 0.0, -> in shadow
  float inLight = sample(tc, r);
  // Clamp angle so that only pixels within the range are in light
  inLight = inLight * step(angle, radiansClamp.y) * step(radiansClamp.x, angle);

  // Multiply the blur amount by the distance from the center
  // So that the blurring increases as distance increases
  float blur = (1.0 / resolution) * smoothstep(0.0, 0.1, r);

  // Use gaussian blur to apply blur effecy
  float sum = 0.0;

  sum += sample(vec2(tc.x - blur * 4.0, tc.y), r) * 0.05;
  sum += sample(vec2(tc.x - blur * 3.0, tc.y), r) * 0.09;
  sum += sample(vec2(tc.x - blur * 2.0, tc.y), r) * 0.12;
  sum += sample(vec2(tc.x - blur * 1.0, tc.y), r) * 0.15;

  sum += inLight * 0.16;

  sum += sample(vec2(tc.x + blur * 1.0, tc.y), r) * 0.15;
  sum += sample(vec2(tc.x + blur * 2.0, tc.y), r) * 0.12;
  sum += sample(vec2(tc.x + blur * 3.0, tc.y), r) * 0.09;
  sum += sample(vec2(tc.x + blur * 4.0, tc.y), r) * 0.05;

  // Mix with the softShadow uniform to toggle degree of softShadows
  float finalLight = mix(inLight, sum, softShadow);

  // Multiply the final light value with the distance, to give a radial falloff
  // Use as the alpha value, with the light colour being the RGB values
  f_colour = vec4(normLightColour, finalLight * smoothstep(1.0, falloff, r));
}
```

Class

The `Rays` class takes in a texture and array of light information, applies the aforementioned shaders, and blends the final result with the original texture.

rays.py

```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.utils.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)
```