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 specific directories and Python files.

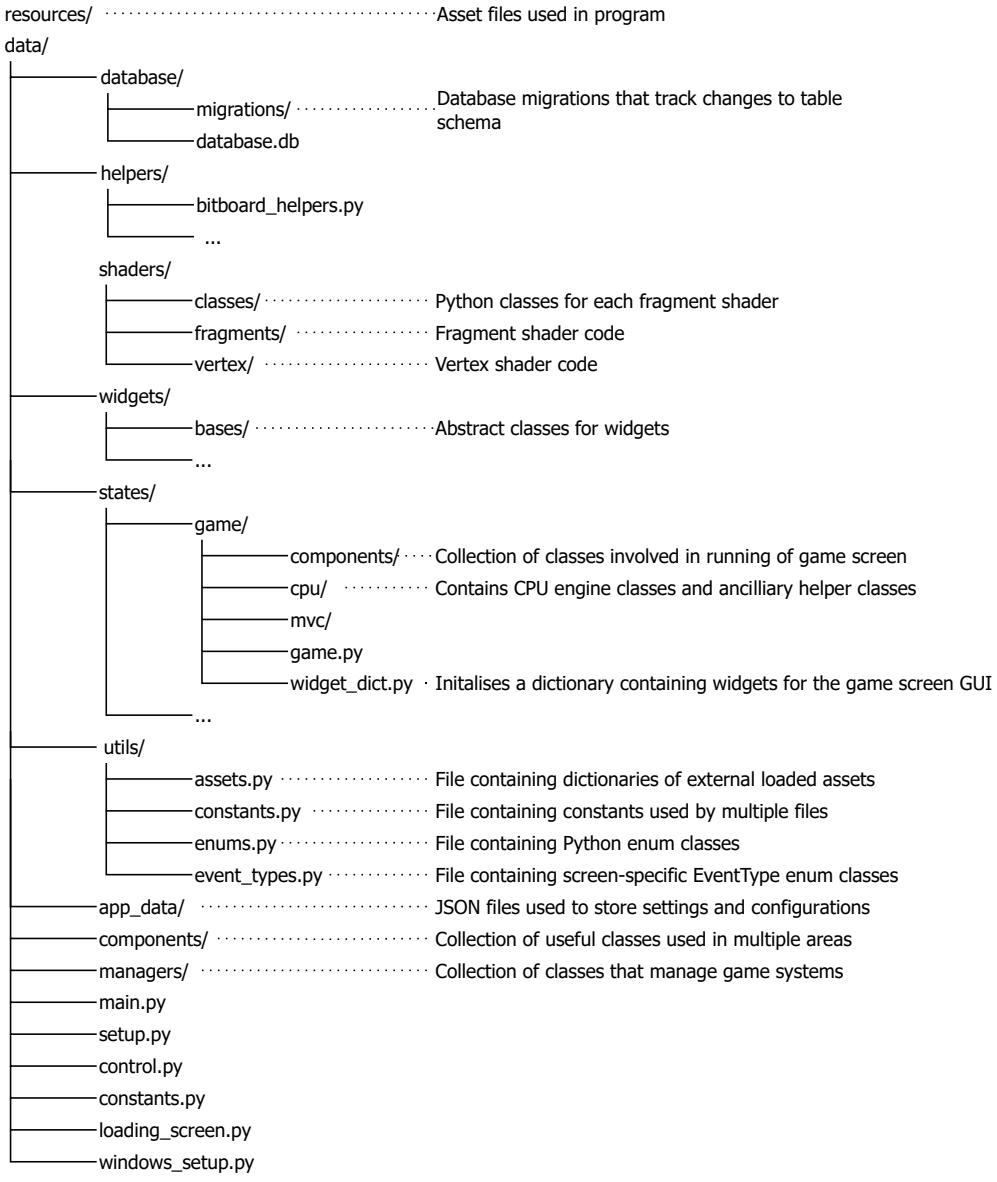

Figure 1.1: File tree diagram

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`

```
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

    # Initialise dictionary containing each screen in the game, referenced in Control class by the current state's 'next' and 'previous' attributes, corresponding to a key in this dictionary
    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):
    """
    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.
72          """
73          for event in pygame.event.get():
74              if event.type == pygame.QUIT:
75                  pygame.quit()
76                  sys.exit()
77
78      def draw(self):
79          """
80          Draws logo to screen.
81          """
82          window.screen.fill((0, 0, 0))
83
84          self._logo_surface.set_alpha(self.logo_opacity)
85          window.screen.blit(self._logo_surface, self.logo_position)
86
87          window.update()
88
89      def run(self):
90          """
91          Runs while the thread is still setting up our screens, or the minimum
    loading screen duration is not reached yet.
92          """
93          while self._thread.is_alive() or self.duration_not_over:
94              self.event_loop()
95              self.draw()
96              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)`

```
1  import pygame
2  from PIL import Image
3  from functools import cache
4  from random import randint
5  import math
6
7  @cache
8  def scale_and_cache(image, target_size):
9      """
10     Caches image when resized repeatedly.
11
12     Args:
13         image (pygame.Surface): Image surface to be resized.
14         target_size (tuple[float, float]): New image size.
15
16     Returns:
17         pygame.Surface: Resized image surface.
18     """
19     return pygame.transform.scale(image, target_size)
20
21 @cache
22 def smoothscale_and_cache(image, target_size):
23     """
24     Same as scale_and_cache, but with the Pygame smoothscale function.
25
26     Args:
27         image (pygame.Surface): Image surface to be resized.
28         target_size (tuple[float, float]): New image size.
```

```
29
30      Returns:
31          pygame.Surface: Resized image surface.
32      """
33      return pygame.transform.smoothscale(image, target_size)
34
35  def gif_to_frames(path):
36      """
37      Uses the PIL library to break down GIFs into individual frames.
38
39      Args:
40          path (str): Directory path to GIF file.
41
42      Yields:
43          PIL.Image: Single frame.
44      """
45      try:
46          image = Image.open(path)
47
48          first_frame = image.copy().convert('RGBA')
49          yield first_frame
50          image.seek(1)
51
52          while True:
53              current_frame = image.copy()
54              yield current_frame
55              image.seek(image.tell() + 1)
56      except EOFError:
57          pass
58
59  def get_perimeter_sample(image_size, number):
60      """
61      Used for particle drawing class, generates roughly equally distributed points
        around a rectangular image surface's perimeter.
62
63      Args:
64          image_size (tuple[float, float]): Image surface size.
65          number (int): Number of points to be generated.
66
67      Returns:
68          list[tuple[int, int], ...]: List of random points on perimeter of image
        surface.
69      """
70      perimeter = 2 * (image_size[0] + image_size[1])
71      # 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
72      perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in
        range(0, number)]
73      pos_list = []
74
75      for perimeter_offset in perimeter_offsets:
76          # For every point, add a random offset
77          max_displacement = int(perimeter / (number * 4))
78          perimeter_offset += randint(-max_displacement, max_displacement)
79
80          if perimeter_offset > perimeter:
81              perimeter_offset -= perimeter
82
83          # Convert 1D distance back into 2D points on image surface perimeter
84          if perimeter_offset < image_size[0]:
85              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.
```

```
147     """
148     return (dest_vertex[0] - src_vertex[0], dest_vertex[1] - src_vertex[1])
149 
150 def get_next_corner(vertex, image_size):
151     """
152     Used in particle drawing system, finds coordinates of the next corner going
        clockwise, given a point on the perimeter.
153 
154     Args:
155         vertex (list[int, int]): Point on perimeter.
156         image_size (list[int, int]): Image size.
157 
158     Returns:
159         list[int, int]: Coordinates of corner on perimeter.
160     """
161     corners = [(0, 0), (image_size[0], 0), (image_size[0], image_size[1]), (0,
        image_size[1])]
162 
163     if vertex in corners:
164         return corners[(corners.index(vertex) + 1) % len(corners)]
165 
166     if vertex[1] == 0:
167         return (image_size[0], 0)
168     elif vertex[0] == image_size[0]:
169         return image_size
170     elif vertex[1] == image_size[1]:
171         return (0, image_size[1])
172     elif vertex[0] == 0:
173         return (0, 0)
174 
175 def pil_image_to_surface(pil_image):
176     """
177     Args:
178         pil_image (PIL.Image): Image to be converted.
179 
180     Returns:
181         pygame.Surface: Converted image surface.
182     """
183     return pygame.image.frombytes(pil_image.tobytes(), pil_image.size, pil_image.
        mode).convert()
184 
185 def calculate_frame_index(elapsed_milliseconds, start_index, end_index, fps):
186     """
187     Determine frame of animated GIF to be displayed.
188 
189     Args:
190         elapsed_milliseconds (int): Milliseconds since GIF started playing.
191         start_index (int): Start frame of GIF.
192         end_index (int): End frame of GIF.
193         fps (int): Number of frames to be played per second.
194 
195     Returns:
196         int: Displayed frame index of GIF.
197     """
198     ms_per_frame = int(1000 / fps)
199     return start_index + ((elapsed_milliseconds // ms_per_frame) % (end_index -
        start_index))
200 
201 def draw_background(screen, background, current_time=0):
202     """
203     Draws background to screen
204 
```

```
    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)

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)

```
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:
```

```
85          radius (float): Radius of circle.
86          colour (pygame.Color): Fill colour.
87          border_colour (pygame.Color): Border colour.
88          border_width (float): Border width.
89
90      Returns:
91          pygame.Surface: Circle surface.
92      """
93      thumb_surface = pygame.Surface((radius * 2, radius * 2), pygame.SRCALPHA)
94      pygame.draw.circle(thumb_surface, border_colour, (radius, radius), radius,
        width=int(border_width))
95      pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius -
        border_width))
96
97      return thumb_surface
98
99  def create_square_gradient(side_length, colour):
100     """
101     Creates a square gradient for the colour picker widget, gradient transitioning
         between saturation and value.
102     Uses smoothscale to blend between colour values for individual pixels.
103
104     Args:
105         side_length (float): Length of a square side.
106         colour (pygame.Color): Colour with desired hue value.
107
108     Returns:
109         pygame.Surface: Square gradient surface.
110     """
111     square_surface = pygame.Surface((side_length, side_length))
112
113     mix_1 = pygame.Surface((1, 2))
114     mix_1.fill((255, 255, 255))
115     mix_1.set_at((0, 1), (0, 0, 0))
116     mix_1 = pygame.transform.smoothscale(mix_1, (side_length, side_length))
117
118     hue = colour.hsva[0]
119     saturated_rgb = pygame.Color(0)
120     saturated_rgb.hsva = (hue, 100, 100)
121
122     mix_2 = pygame.Surface((2, 1))
123     mix_2.fill((255, 255, 255))
124     mix_2.set_at((1, 0), saturated_rgb)
125     mix_2 = pygame.transform.smoothscale(mix_2,(side_length, side_length))
126
127     mix_1.blit(mix_2, (0, 0), special_flags=pygame.BLEND_MULT)
128
129     square_surface.blit(mix_1, (0, 0))
130
131     return square_surface
132
133 def create_switch(size, colour):
134     """
135     Creates surface for switch toggle widget.
136
137     Args:
138         size (list[int, int]): Image size.
139         colour (pygame.Color): Fill colour.
140
141     Returns:
142         pygame.Surface: Switch surface.
143     """
```

```
144     switch_surface = pygame.Surface((size[0], size[1]), pygame.SRCALPHA)
145     pygame.draw.rect(switch_surface, colour, (0, 0, size[0], size[1]),
        border_radius=int(size[1] / 2))
146 
147     return switch_surface
148 
149 def create_text_box(size, border_width, colours):
150     """
151     Creates bordered textbox with shadow, flat, and highlighted vertical regions.
152 
153     Args:
154         size (list[int, int]): Image size.
155         border_width (float): Border width.
156         colours (list[pygame.Color, ...]): List of 4 colours, representing border
        colour, shadow colour, flat colour and highlighted colour.
157 
158     Returns:
159         pygame.Surface: Textbox surface.
160     """
161     surface = pygame.Surface(size, pygame.SRCALPHA)
162 
163     pygame.draw.rect(surface, colours[0], (0, 0, *size))
164     pygame.draw.rect(surface, colours[2], (border_width, border_width, size[0] - 2
         * border_width, size[1] - 2 * border_width))
165     pygame.draw.rect(surface, colours[3], (border_width, border_width, size[0] - 2
         * border_width, border_width))
166     pygame.draw.rect(surface, colours[1], (border_width, size[1] - 2 *
        border_width, size[0] - 2 * border_width, border_width))
167 
168     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

```
1 from data.helpers.data_helpers import get_themes, get_user_settings
2 
3 themes = get_themes()
4 user_settings = get_user_settings()
5 
6 def flatten_dictionary_generator(dictionary, parent_key=None):
7     """
8     Recursive depth-first search to yield all items in a dictionary.
9 
10     Args:
11         dictionary (dict): Dictionary to be iterated through.
12         parent_key (str, optional): Prefix added to every key. Defaults to None.
13 
14     Yields:
15         dict | tuple[str, str]: Another dictionary or key, value pair.
16     """
17     for key, value in dictionary.items():
18         if parent_key:
19             new_key = parent_key + key.capitalize()
20         else:
21             new_key = key
```

```
22
23          if isinstance(value, dict):
24              yield from flatten_dictionary(value, new_key).items()
25          else:
26              yield new_key, value
27
28  def flatten_dictionary(dictionary, parent_key=''):
29      return dict(flatten_dictionary_generator(dictionary, parent_key))
30
31  class ThemeManager:
32      def __init__(self):
33          self.__dict__.update(flatten_dictionary(themes['colours']))
34          self.__dict__.update(flatten_dictionary(themes['dimensions']))
35
36      def __getitem__(self, arg):
37          """
38          Override default class's __getitem__ dunder method, to make retrieving an
        instance attribute nicer with [] notation.
39
40          Args:
41              arg (str): Attribute name.
42
43          Raises:
44              KeyError: Instance does not have requested attribute.
45
46          Returns:
47              str | int: Instance attribute.
48          """
49          item = self.__dict__.get(arg)
50
51          if item is None:
52              raise KeyError('(ThemeManager.__getitem__) Requested theme item not
        found:', arg)
53
54          return item
55
56  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

```
1  import pygame
2  from data.helpers.board_helpers import coords_to_screen_pos
3  from data.utils.enums import LaserType, Colour, ShaderType
4  from data.managers.animation import animation
5  from data.utils.assets import GRAPHICS, SFX
6  from data.utils.constants import EMPTY_BB
7  from data.managers.window import window
8  from data.managers.audio import audio
9
10 type_to_image = {
11     LaserType.END: ['laser_end_1', 'laser_end_2'],
12     LaserType.STRAIGHT: ['laser_straight_1', 'laser_straight_2'],
13     LaserType.CORNER: ['laser_corner_1', 'laser_corner_2']
14 }
15
```

```
16 GLOW_SCALE_FACTOR = 1.5
17
18 class LaserDraw:
19     def __init__(self, board_position, board_size):
20         self._board_position = board_position
21         self._square_size = board_size[0] / 10
22         self._laser_lists = []
23
24     @property
25     def firing(self):
26         return len(self._laser_lists) > 0
27
28     def add_laser(self, laser_result, laser_colour):
29         """
30         Adds a laser to the board.
31
32         Args:
33             laser_result (Laser): Laser class instance containing laser trajectory
       info.
34             laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
35         """
36         laser_path = laser_result.laser_path.copy()
37         laser_types = [LaserType.END]
38         # List of angles in degree to rotate the laser image surface when drawn
39         laser_rotation = [laser_path[0][1]]
40         laser_lights = []
41
42         # Iterates through every square laser passes through
43         for i in range(1, len(laser_path)):
44             previous_direction = laser_path[i-1][1]
45             current_coords, current_direction = laser_path[i]
46
47             if current_direction == previous_direction:
48                 laser_types.append(LaserType.STRAIGHT)
49                 laser_rotation.append(current_direction)
50             elif current_direction == previous_direction.get_clockwise():
51                 laser_types.append(LaserType.CORNER)
52                 laser_rotation.append(current_direction)
53             elif current_direction == previous_direction.get_anticlockwise():
54                 laser_types.append(LaserType.CORNER)
55                 laser_rotation.append(current_direction.get_anticlockwise())
56
57             # Adds a shader ray effect on the first and last square of the laser
     trajectory
58             if i in [1, len(laser_path) - 1]:
59                 abs_position = coords_to_screen_pos(current_coords, self.
     _board_position, self._square_size)
60                 laser_lights.append([
61                     (abs_position[0] / window.size[0], abs_position[1] / window.
     size[1]),
62                     0.35,
63                     (0, 0, 255) if laser_colour == Colour.BLUE else (255, 0, 0),
64                 ])
65
66         # Sets end laser draw type if laser hits a piece or piece is anubis
67         if laser_result.end_cap:
68             laser_types[-1] = LaserType.END
69             laser_path[-1] = (laser_path[-1][0], laser_path[-2][1].get_opposite())
70             laser_rotation[-1] = laser_path[-2][1].get_opposite()
71
72         # Played audio cue if piece is destroyed
73         if laser_result.hit_square_bitboard != EMPTY_BB:
```

```
74              audio.play_sfx(SFX['piece_destroy'])
75
76          laser_path = [(coords, rotation, type) for (coords, dir), rotation, type
    in zip(laser_path, laser_rotation, laser_types)]
77          self._laser_lists.append((laser_path, laser_colour))
78
79          window.clear_effect(ShaderType.RAYS)
80          window.set_effect(ShaderType.RAYS, lights=laser_lights)
81          animation.set_timer(1000, self.remove_laser)
82
83          audio.play_sfx(SFX['laser_1'])
84          audio.play_sfx(SFX['laser_2'])
85
86      def remove_laser(self):
87          """
88          Removes a laser from the board.
89          """
90          self._laser_lists.pop(0)
91
92          if len(self._laser_lists) == 0:
93              window.clear_effect(ShaderType.RAYS)
94
95      def draw_laser(self, screen, laser_list, glow=True):
96          """
97          Draws every laser on the screen.
98
99          Args:
100             screen (pygame.Surface): The screen to draw on.
101             laser_list (list): The list of laser segments to draw.
102             glow (bool, optional): Whether to draw a glow effect. Defaults to True
    .
103         """
104         laser_path, laser_colour = laser_list
105         laser_list = []
106         glow_list = []
107
108         for coords, rotation, type in laser_path:
109             square_x, square_y = coords_to_screen_pos(coords, self._board_position
    , self._square_size)
110             image = GRAPHICS[type_to_image[type][laser_colour]]
111             rotated_image = pygame.transform.rotate(image, rotation.to_angle())
112             scaled_image = pygame.transform.scale(rotated_image, (self.
    _square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating
    black lines
113             laser_list.append((scaled_image, (square_x, square_y)))
114
115             # Scales up the laser image surface as a glow surface
116             scaled_glow = pygame.transform.scale(rotated_image, (self._square_size
     * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
117             offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
118             glow_list.append((scaled_glow, (square_x - offset, square_y - offset))
    )
119
120         # Scaled glow surfaces drawn on top with the RGB_ADD blend mode
121         if glow:
122             screen.fblits(glow_list, pygame.BLEND_RGB_ADD)
123
124         screen.blits(laser_list)
125
126     def draw(self, screen):
127         """
128         Draws all lasers on the screen.
```

```
129
130         Args:
131             screen (pygame.Surface): The screen to draw on.
132         """
133         for laser_list in self._laser_lists:
134             self.draw_laser(screen, laser_list)
135
136     def handle_resize(self, board_position, board_size):
137         """
138         Handles resizing of the board.
139
140         Args:
141             board_position (tuple[int, int]): The new position of the board.
142             board_size (tuple[int, int]): The new size of the board.
143         """
144         self._board_position = board_position
145         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

```
1  import pygame
2  from random import randint
3  from data.helpers.asset_helpers import get_perimeter_sample, get_vector,
       get_angle_between_vectors, get_next_corner
4  from data.states.game.components.piece_sprite import PieceSprite
5  from data.helpers.data_helpers import get_user_settings
6
7  particles_disabled = not(get_user_settings()['particles'])
8
9  class ParticlesDraw:
10     def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
11         self._particles = []
12         self._glow_particles = []
13
14         self._gravity = gravity
15         self._rotation = rotation
16         self._shrink = shrink
17         self._opacity = opacity
18
19     def fragment_image(self, image, number):
20         image_size = image.get_rect().size
21         """
22         1. Takes an image surface and samples random points on the perimeter.
23         2. Iterates through points, and depending on the nature of two consecutive
        points, finds a corner between them.
24         3. Draws a polygon with the points as the vertices to mask out the area
       not in the fragment.
25
26         Args:
27             image (pygame.Surface): Image to fragment.
28             number (int): The number of fragments to create.
29
30         Returns:
31             list[pygame.Surface]: List of image surfaces with fragment of original
        surface drawn on top.
```

```
        """
        center = image.get_rect().center
        points_list = get_perimeter_sample(image_size, number)
        fragment_list = []

        points_list.append(points_list[0])

        # Iterate through points_list, using the current point and the next one
        for i in range(len(points_list) - 1):
            vertex_1 = points_list[i]
            vertex_2 = points_list[i + 1]
            vector_1 = get_vector(center, vertex_1)
            vector_2 = get_vector(center, vertex_2)
            angle = get_angle_between_vectors(vector_1, vector_2)

            cropped_image = pygame.Surface(image_size, pygame.SRCALPHA)
            cropped_image.fill((0, 0, 0, 0))
            cropped_image.blit(image, (0, 0))

            corners_to_draw = None

            if vertex_1[0] == vertex_2[0] or vertex_1[1] == vertex_2[1]: # Points on the same side
                corners_to_draw = 4

            elif abs(vertex_1[0] - vertex_2[0]) == image_size[0] or abs(vertex_1[1] - vertex_2[1]) == image_size[1]: # Points on opposite sides
                corners_to_draw = 2

            elif angle < 180: # Points on adjacent sides
                corners_to_draw = 3

            else:
                corners_to_draw = 1

            corners_list = []
            for j in range(corners_to_draw):
                if len(corners_list) == 0:
                    corners_list.append(get_next_corner(vertex_2, image_size))
                else:
                    corners_list.append(get_next_corner(corners_list[-1], image_size))

            pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *corners_list, vertex_1))

            fragment_list.append(cropped_image)

        return fragment_list

    def add_captured_piece(self, piece, colour, rotation, position, size):
        """
        Adds a captured piece to fragment into particles.

        Args:
            piece (Piece): The piece type.
            colour (Colour): The active colour of the piece.
            rotation (int): The rotation of the piece.
            position (tuple[int, int]): The position where particles originate from.
            size (tuple[int, int]): The size of the piece.
        """
```

```
89          if particles_disabled:
90              return
91
92          piece_sprite = PieceSprite(piece, colour, rotation)
93          piece_sprite.set_geometry((0, 0), size)
94          piece_sprite.set_image()
95
96          particles = self.fragment_image(piece_sprite.image, 5)
97
98          for particle in particles:
99              self.add_particle(particle, position)
100
101     def add_sparks(self, radius, colour, position):
102         """
103         Adds laser spark particles.
104
105         Args:
106             radius (int): The radius of the sparks.
107             colour (Colour): The active colour of the sparks.
108             position (tuple[int, int]): The position where particles originate
    from.
109         """
110         if particles_disabled:
111             return
112
113         for i in range(randint(10, 15)):
114             velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
115             random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in
    colour]
116             self._particles.append([None, [radius, random_colour], [*position],
    velocity, 0])
117
118     def add_particle(self, image, position):
119         """
120         Adds a particle.
121
122         Args:
123             image (pygame.Surface): The image of the particle.
124             position (tuple): The position of the particle.
125         """
126         if particles_disabled:
127             return
128
129         velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
130
131         # Each particle is stored with its attributes: [surface, copy of surface,
    position, velocity, lifespan]
132         self._particles.append([image, image.copy(), [*position], velocity, 0])
133
134     def update(self):
135         """
136         Updates each particle and its attributes.
137         """
138         for i in range(len(self._particles) - 1, -1, -1):
139             particle = self._particles[i]
140
141             #update position
142             particle[2][0] += particle[3][0]
143             particle[2][1] += particle[3][1]
144
145             #update lifespan
146             self._particles[i][4] += 0.01
```

```
147
148             if self._particles[i][4] >= 1:
149                 self._particles.pop(i)
150                 continue
151
152             if isinstance(particle[1], pygame.Surface): # Particle is a piece
153                 # Update velocity
154                 particle[3][1] += self._gravity
155
156                 # Update size
157                 image_size = particle[1].get_rect().size
158                 end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink)
     * image_size[1])
159                 target_size = (image_size[0] - particle[4] * (image_size[0] -
    end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))
160
161                 # Update rotation
162                 rotation = (self._rotation if particle[3][0] <= 0 else -self.
    _rotation) * particle[4]
163
164                 updated_image = pygame.transform.scale(pygame.transform.rotate(
    particle[1], rotation), target_size)
165
166             elif isinstance(particle[1], list): # Particle is a spark
167                 # Update size
168                 end_radius = (1 - self._shrink) * particle[1][0]
169                 target_radius = particle[1][0] - particle[4] * (particle[1][0] -
    end_radius)
170
171                 updated_image = pygame.Surface((target_radius * 2, target_radius *
     2), pygame.SRCALPHA)
172                 pygame.draw.circle(updated_image, particle[1][1], (target_radius,
    target_radius), target_radius)
173
174             # Update opacity
175             alpha = 255 - particle[4] * (255 - self._opacity)
176
177             updated_image.fill((255, 255, 255, alpha), None, pygame.
    BLEND_RGBA_MULT)
178
179             particle[0] = updated_image
180
181     def draw(self, screen):
182         """
183         Draws the particles, indexing the surface and position attributes for each
     particle.
184
185         Args:
186             screen (pygame.Surface): The screen to draw on.
187         """
188         screen.blits([
189             (particle[0], particle[2]) for particle in self._particles
190         ])
```

1.4.3 Widget Bases

Widget bases are used as the base classes 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 overridden. 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']

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
```

```
43          self._raw_surface_size = DEFAULT_SURFACE_SIZE
44
45          self._parent = kwargs.get('parent')
46
47          self._relative_font_size = None # Set in subclass
48
49          self._relative_position = kwargs.get('relative_position')
50          self._relative_margin = theme['margin'] / self._raw_surface_size[1]
51          self._relative_border_width = theme['borderWidth'] / self.
    _raw_surface_size[1]
52          self._relative_border_radius = theme['borderRadius'] / self.
    _raw_surface_size[1]
53
54          self._border_colour = pygame.Color(theme['borderPrimary'])
55          self._text_colour = pygame.Color(theme['textPrimary'])
56          self._fill_colour = pygame.Color(theme['fillPrimary'])
57          self._font = DEFAULT_FONT
58
59          self._anchor_x = kwargs.get('anchor_x') or 'left'
60          self._anchor_y = kwargs.get('anchor_y') or 'top'
61          self._fixed_position = kwargs.get('fixed_position')
62          scale_mode = kwargs.get('scale_mode') or 'both'
63
64          if kwargs.get('relative_size'):
65              match scale_mode:
66                  case 'height':
67                      self._relative_size = kwargs.get('relative_size')
68                  case 'width':
69                      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])
70                  case 'both':
71                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
72                  case _:
73                      raise ValueError('(_Widget.__init__) Unknown scale mode:',
    scale_mode)
74          else:
75              self._relative_size = (1, 1)
76
77          if 'margin' in kwargs:
78              self._relative_margin = kwargs.get('margin') / self._raw_surface_size
    [1]
79
80              if (self._relative_margin * 2) > min(self._relative_size[0], self.
    _relative_size[1]):
81                  raise ValueError('(_Widget.__init__) Margin larger than specified
    size!')
82
83          if 'border_width' in kwargs:
84              self._relative_border_width = kwargs.get('border_width') / self.
    _raw_surface_size[1]
85
86          if 'border_radius' in kwargs:
87              self._relative_border_radius = kwargs.get('border_radius') / self.
    _raw_surface_size[1]
88
89          if 'border_colour' in kwargs:
90              self._border_colour = pygame.Color(kwargs.get('border_colour'))
91
92          if 'fill_colour' in kwargs:
93              self._fill_colour = pygame.Color(kwargs.get('fill_colour'))
```

```
94
95          if 'text_colour' in kwargs:
96              self._text_colour = pygame.Color(kwargs.get('text_colour'))
97
98          if 'font' in kwargs:
99              self._font = kwargs.get('font')
100
101     @property
102     def surface_size(self):
103         """
104         Gets the size of the surface widget is drawn on.
105         Can be either the window size, or another widget size if assigned to a
    parent.
106
107         Returns:
108             tuple[int, int]: The size of the surface.
109         """
110         if self._parent:
111             return self._parent.size
112         else:
113             return self._raw_surface_size
114
115     @property
116     def position(self):
117         """
118         Gets the position of the widget.
119         Accounts for fixed position attribute, where widget is positioned in
    pixels regardless of screen size.
120         Acounts for anchor direction, where position attribute is calculated
    relative to one side of the screen.
121
122         Returns:
123             tuple[int, int]: The position of the widget.
124         """
125         x, y = None, None
126         if self._fixed_position:
127             x, y = self._fixed_position
128         if x is None:
129             x = self._relative_position[0] * self.surface_size[0]
130         if y is None:
131             y = self._relative_position[1] * self.surface_size[1]
132
133         if self._anchor_x == 'left':
134             x = x
135         elif self._anchor_x == 'right':
136             x = self.surface_size[0] - x - self.size[0]
137         elif self._anchor_x == 'center':
138             x = (self.surface_size[0] / 2 - self.size[0] / 2) + x
139
140         if self._anchor_y == 'top':
141             y = y
142         elif self._anchor_y == 'bottom':
143             y = self.surface_size[1] - y - self.size[1]
144         elif self._anchor_y == 'center':
145             y = (self.surface_size[1] / 2 - self.size[1] / 2) + y
146
147         # Position widget relative to parent, if exists.
148         if self._parent:
149             return (x + self._parent.position[0], y + self._parent.position[1])
150
151         return (x, y)
152
```

```
153     @property
154     def size(self):
155         return (self._relative_size[0] * self.surface_size[1], self._relative_size
        [1] * self.surface_size[1])
156 
157     @property
158     def margin(self):
159         return self._relative_margin * self._raw_surface_size[1]
160 
161     @property
162     def border_width(self):
163         return self._relative_border_width * self._raw_surface_size[1]
164 
165     @property
166     def border_radius(self):
167         return self._relative_border_radius * self._raw_surface_size[1]
168 
169     @property
170     def font_size(self):
171         return self._relative_font_size * self.surface_size[1]
172 
173     def set_image(self):
174         """
175         Abstract method to draw widget.
176         """
177         raise NotImplementedError
178 
179     def set_geometry(self):
180         """
181         Sets the position and size of the widget.
182         """
183         self.rect = self.image.get_rect()
184 
185         if self._anchor_x == 'left':
186             if self._anchor_y == 'top':
187                 self.rect.topleft = self.position
188             elif self._anchor_y == 'bottom':
189                 self.rect.topleft = self.position
190             elif self._anchor_y == 'center':
191                 self.rect.topleft = self.position
192         elif self._anchor_x == 'right':
193             if self._anchor_y == 'top':
194                 self.rect.topleft = self.position
195             elif self._anchor_y == 'bottom':
196                 self.rect.topleft = self.position
197             elif self._anchor_y == 'center':
198                 self.rect.topleft = self.position
199         elif self._anchor_x == 'center':
200             if self._anchor_y == 'top':
201                 self.rect.topleft = self.position
202             elif self._anchor_y == 'bottom':
203                 self.rect.topleft = self.position
204             elif self._anchor_y == 'center':
205                 self.rect.topleft = self.position
206 
207     def set_surface_size(self, new_surface_size):
208         """
209         Sets the new size of the surface widget is drawn on.
210 
211         Args:
212             new_surface_size (tuple[int, int]): The new size of the surface.
213         """
```

```
214         self._raw_surface_size = new_surface_size
215
216     def process_event(self, event):
217         """
218         Abstract method to handle events.
219
220         Args:
221             event (pygame.Event): The event to process.
222         """
223         raise NotImplementedError
```

Circular

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

```
1  from data.components.circular_linked_list import CircularLinkedList
2
3  class _Circular:
4      def __init__(self, items_dict, **kwargs):
5          # The key, value pairs are stored within a dictionary, while the keys to
       access them are stored within circular linked list.
6          self._items_dict = items_dict
7          self._keys_list = CircularLinkedList(list(items_dict.keys()))
8
9      @property
10     def current_key(self):
11         """
12         Gets the current head node of the linked list, and returns a key stored as
        the node data.
13         Returns:
14             Data of linked list head.
15         """
16         return self._keys_list.get_head().data
17
18     @property
19     def current_item(self):
20         """
21         Gets the value in self._items_dict with the key being self.current_key.
22
23         Returns:
24             Value stored with key being current head of linked list.
25         """
26         return self._items_dict[self.current_key]
27
28     def set_next_item(self):
29         """
30         Sets the next item in as the current item.
31         """
32         self._keys_list.shift_head()
33
34     def set_previous_item(self):
35         """
36         Sets the previous item as the current item.
37         """
38         self._keys_list.unshift_head()
39
40     def set_to_key(self, key):
41         """
42         Sets the current item to the specified key.
43
```

```
        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

```
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
```

```
158
159             current_node = current_node.next
160             if current_node == self._head:
161                 return False
162
163     def shift_head(self):
164         """
165         Shifts the head of the circular linked list to the next node.
166         """
167         self._head = self._head.next
168
169     def unshift_head(self):
170         """
171         Shifts the head of the circular linked list to the previous node.
172         """
173         self._head = self._head.previous
174
175     def get_head(self):
176         """
177         Gets the head node of the circular linked list.
178
179         Returns:
180             Node: The head node.
181         """
182         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

```
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.

```

```
48          Returns:
49              CustomEvent: Initialised CustomEvent instance.
50          """
51          if event_type in required_args:
52
53              for required_arg in required_args[event_type]:
54                  if required_arg not in kwargs:
55                      raise ValueError(f"Argument '{required_arg}' required for {
    event_type.name} event (GameEvent.create_event)")
56
57              for kwarg in kwargs:
58                  if kwarg not in required_args[event_type]:
59                      raise ValueError(f"Argument '{kwarg}' not included in
    required_args dictionary for event '{event_type}'! (GameEvent.create_event)")
60
61              return event_cls(event_type, **kwargs)
62
63          else:
64              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

```
1  from data.widgets.reactive_button import ReactiveButton
2  from data.utils.constants import WidgetState
3  from data.widgets.icon import Icon
4
5  class ReactiveIconButton(ReactiveButton):
6      def __init__(self, base_icon, hover_icon, press_icon, **kwargs):
7          # Composition is used here, to initialise the Icon widgets for each widget
       state
8          widgets_dict = {
9              WidgetState.BASE: Icon(
10                 parent=kwargs.get('parent'),
11                 relative_size=kwargs.get('relative_size'),
12                 relative_position=(0, 0),
13                 icon=base_icon,
14                 fill_colour=(0, 0, 0, 0),
15                 border_width=0,
16                 margin=0,
17                 fit_icon=True,
18             ),
19             WidgetState.HOVER: Icon(
20                 parent=kwargs.get('parent'),
21                 relative_size=kwargs.get('relative_size'),
22                 relative_position=(0, 0),
23                 icon=hover_icon,
24                 fill_colour=(0, 0, 0, 0),
25                 border_width=0,
26                 margin=0,
27                 fit_icon=True,
28             ),
29             WidgetState.PRESS: Icon(
30                 parent=kwargs.get('parent'),
31                 relative_size=kwargs.get('relative_size'),
32                 relative_position=(0, 0),
33                 icon=press_icon,
34                 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

```
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
```

```
41
42      def set_image(self):
43          """
44          Sets current icon to image.
45          """
46          self.current_item.set_image()
47          self.image = self.current_item.image
48
49      def set_geometry(self):
50          """
51          Sets size and position of widget.
52          """
53          super().set_geometry()
54          self.current_item.set_geometry()
55          self.current_item.rect.topleft = self.rect.topleft
56
57      def set_surface_size(self, new_surface_size):
58          """
59          Overrides base method to resize every widget state icon, not just the
    current one.
60
61          Args:
62              new_surface_size (list[int, int]): New surface size.
63          """
64          super().set_surface_size(new_surface_size)
65          for item in self._items_dict.values():
66              item.set_surface_size(new_surface_size)
67
68      def process_event(self, event):
69          """
70          Processes Pygame events.
71
72          Args:
73              event (pygame.Event): Event to process.
74
75          Returns:
76              CustomEvent: CustomEvent of current item, with current key included
77          """
78          widget_event = super().process_event(event)
79          self.current_item.process_event(event)
80
81          if widget_event:
82              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

```
1  import pygame
2  from data.helpers.widget_helpers import create_slider_gradient
3  from data.helpers.asset_helpers import smoothscale_and_cache
4  from data.widgets.slider_thumb import _SliderThumb
5  from data.widgets.bases.widget import _Widget
6  from data.utils.constants import WidgetState
7
8  class _ColourSlider(_Widget):
9      def __init__(self, relative_width, **kwargs):
10         super().__init__(relative_size=(relative_width, relative_width * 0.2), **
       kwargs)
```

```
11
12          # Initialise slider thumb.
13          self._thumb = _SliderThumb(radius=self.size[1] / 2, border_colour=self.
        _border_colour)
14
15          self._selected_percent = 0
16          self._last_mouse_x = None
17
18          self._gradient_surface = create_slider_gradient(self.gradient_size, self.
        border_width, self._border_colour)
19          self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)
20
21      @property
22      def gradient_size(self):
23          return (self.size[0] - 2 * (self.size[1] / 2), self.size[1] / 2)
24
25      @property
26      def gradient_position(self):
27          return (self.size[1] / 2, self.size[1] / 4)
28
29      @property
30      def thumb_position(self):
31          return (self.gradient_size[0] * self._selected_percent, 0)
32
33      @property
34      def selected_colour(self):
35          colour = pygame.Color(0)
36          colour.hsva = (int(self._selected_percent * 360), 100, 100, 100)
37          return colour
38
39      def calculate_gradient_percent(self, mouse_pos):
40          """
41          Calculate what percentage slider thumb is at based on change in mouse
        position.
42
43          Args:
44              mouse_pos (list[int, int]): Position of mouse on window screen.
45
46          Returns:
47              float: Slider scroll percentage.
48          """
49          if self._last_mouse_x is None:
50              return
51
52          x_change = (mouse_pos[0] - self._last_mouse_x) / (self.gradient_size[0] -
        2 * self.border_width)
53          return max(0, min(self._selected_percent + x_change, 1))
54
55      def relative_to_global_position(self, position):
56          """
57          Transforms position from being relative to widget rect, to window screen.
58
59          Args:
60              position (list[int, int]): Position relative to widget rect.
61
62          Returns:
63              list[int, int]: Position relative to window screen.
64          """
65          relative_x, relative_y = position
66          return (relative_x + self.position[0], relative_y + self.position[1])
67
68      def set_colour(self, new_colour):
```

```
69          """
70          Sets selected_percent based on the new colour's hue.
71
72          Args:
73              new_colour (pygame.Color): New slider colour.
74          """
75          colour = pygame.Color(new_colour)
76          hue = colour.hsva[0]
77          self._selected_percent = hue / 360
78          self.set_image()
79
80      def set_image(self):
81          """
82          Draws colour slider to widget image.
83          """
84          # Scales initalised gradient surface instead of redrawing it everytime
    set_image is called
85          gradient_scaled = smoothscale_and_cache(self._gradient_surface, self.
    gradient_size)
86
87          self.image = pygame.transform.scale(self._empty_surface, (self.size))
88          self.image.blit(gradient_scaled, self.gradient_position)
89
90          # Resets thumb colour, image and position, then draws it to the widget
    image
91          self._thumb.initialise_new_colours(self.selected_colour)
92          self._thumb.set_surface(radius=self.size[1] / 2, border_width=self.
    border_width)
93          self._thumb.set_position(self.relative_to_global_position((self.
    thumb_position[0], self.thumb_position[1])))
94
95          thumb_surface = self._thumb.get_surface()
96          self.image.blit(thumb_surface, self.thumb_position)
97
98      def process_event(self, event):
99          """
100         Processes Pygame events.
101
102         Args:
103             event (pygame.Event): Event to process.
104
105         Returns:
106             pygame.Color: Current colour slider is displaying.
107         """
108         if event.type not in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONDOWN, pygame.
    MOUSEBUTTONUP]:
109             return
110
111         # Gets widget state before and after event is processed by slider thumb
112         before_state = self._thumb.state
113         self._thumb.process_event(event)
114         after_state = self._thumb.state
115
116         # If widget state changes (e.g. hovered -> pressed), redraw widget
117         if before_state != after_state:
118             self.set_image()
119
120         if event.type == pygame.MOUSEMOTION:
121             if self._thumb.state == WidgetState.PRESS:
122                 # Recalculates slider colour based on mouse position change
123                 selected_percent = self.calculate_gradient_percent(event.pos)
124                 self._last_mouse_x = event.pos[0]
```

```
125
126                 if selected_percent is not None:
127                     self._selected_percent = selected_percent
128
129                     return self.selected_colour
130
131         if event.type == pygame.MOUSEBUTTONUP:
132             # When user stops scrolling, return new slider colour
133             self._last_mouse_x = None
134             return self.selected_colour
135
136         if event.type == pygame.MOUSEBUTTONDOWN or before_state != after_state:
137             # Redraws widget when slider thumb is hovered or pressed
138             return self.selected_colour
```

TextInput

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

```
1  import pyperclip
2  import pygame
3  from data.utils.constants import WidgetState, INPUT_COLOURS
4  from data.components.custom_event import CustomEvent
5  from data.widgets.bases.pressable import _Pressable
6  from data.managers.logs import initialise_logger
7  from data.managers.animation import animation
8  from data.widgets.bases.box import _Box
9  from data.utils.enums import CursorMode
10 from data.managers.cursor import cursor
11 from data.managers.theme import theme
12 from data.widgets.text import Text
13
14 logger = initialise_logger(__name__)
15
16 class TextInput(_Box, _Pressable, Text):
17     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):
18         self._cursor_index = None
19         # Multiple inheritance used here, adding the functionality of pressing,
       and custom box colours, to the text widget
20         _Box.__init__(self, box_colours=INPUT_COLOURS)
21         _Pressable.__init__(
22             self,
23             event=None,
24             hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
25             down_func=lambda: self.set_state_colour(WidgetState.PRESS),
26             up_func=lambda: self.set_state_colour(WidgetState.BASE),
27             sfx=None
28         )
29         Text.__init__(self, text="", center=False, box_colours=INPUT_COLOURS[
       WidgetState.BASE], **kwargs)
30
31         self.initialise_new_colours(self._fill_colour)
32         self.set_state_colour(WidgetState.BASE)
33
34         pygame.key.set_repeat(500, 50)
35
36         self._blinking_fps = 1000 / blinking_interval
37         self._cursor_colour = cursor_colour
```

```
38          self._cursor_colour_copy = cursor_colour
39          self._placeholder_colour = placeholder_colour
40          self._text_colour_copy = self._text_colour
41
42          self._placeholder_text = placeholder
43          self._is_placeholder = None
44          if default:
45              self._text = default
46              self.is_placeholder = False
47          else:
48              self._text = self._placeholder_text
49              self.is_placeholder = True
50
51          self._event = event
52          self._validator = validator
53          self._blinking_cooldown = 0
54
55          self._empty_cursor = pygame.Surface((0, 0), pygame.SRCALPHA)
56
57          self.resize_text()
58          self.set_image()
59          self.set_geometry()
60
61      @property
62      # Encapsulated getter method
63      def is_placeholder(self):
64          return self._is_placeholder
65
66      @is_placeholder.setter
67      # Encapsulated setter method, used to replace text colour if placeholder text
        is shown
68      def is_placeholder(self, is_true):
69          self._is_placeholder = is_true
70
71          if is_true:
72              self._text_colour = self._placeholder_colour
73          else:
74              self._text_colour = self._text_colour_copy
75
76      @property
77      def cursor_size(self):
78          cursor_height = (self.size[1] - self.border_width * 2) * 0.75
79          return (cursor_height * 0.1, cursor_height)
80
81      @property
82      def cursor_position(self):
83          current_width = (self.margin / 2)
84          for index, metrics in enumerate(self._font.get_metrics(self._text, size=
        self.font_size)):
85              if index == self._cursor_index:
86                  return (current_width - self.cursor_size[0], (self.size[1] - self.
        cursor_size[1]) / 2)
87
88              glyph_width = metrics[4]
89              current_width += glyph_width
90          return (current_width - self.cursor_size[0], (self.size[1] - self.
        cursor_size[1]) / 2)
91
92      @property
93      def text(self):
94          if self.is_placeholder:
95              return ''
```

```
96
97          return self._text
98
99      def relative_x_to_cursor_index(self, relative_x):
100         """
101         Calculates cursor index using mouse position relative to the widget
    position.
102
103         Args:
104             relative_x (int): Horizontal distance of the mouse from the left side
    of the widget.
105
106         Returns:
107             int: Cursor index.
108         """
109         current_width = 0
110
111         for index, metrics in enumerate(self._font.get_metrics(self._text, size=
    self.font_size)):
112             glyph_width = metrics[4]
113
114             if current_width >= relative_x:
115                 return index
116
117             current_width += glyph_width
118
119         return len(self._text)
120
121     def set_cursor_index(self, mouse_pos):
122         """
123         Sets cursor index based on mouse position.
124
125         Args:
126             mouse_pos (list[int, int]): Mouse position relative to window screen.
127         """
128         if mouse_pos is None:
129             self._cursor_index = mouse_pos
130             return
131
132         relative_x = mouse_pos[0] - (self.margin / 2) - self.rect.left
133         relative_x = max(0, relative_x)
134         self._cursor_index = self.relative_x_to_cursor_index(relative_x)
135
136     def focus_input(self, mouse_pos):
137         """
138         Draws cursor and sets cursor index when user clicks on widget.
139
140         Args:
141             mouse_pos (list[int, int]): Mouse position relative to window screen.
142         """
143         if self.is_placeholder:
144             self._text = ''
145             self.is_placeholder = False
146
147         self.set_cursor_index(mouse_pos)
148         self.set_image()
149         cursor.set_mode(CursorMode.IBEAM)
150
151     def unfocus_input(self):
152         """
153         Removes cursor when user unselects widget.
154         """
```

```
155         if self._text == '':
156             self._text = self._placeholder_text
157             self.is_placeholder = True
158             self.resize_text()
159 
160         self.set_cursor_index(None)
161         self.set_image()
162         cursor.set_mode(CursorMode.ARROW)
163 
164     def set_text(self, new_text):
165         """
166         Called by a state object to change the widget text externally.
167 
168         Args:
169             new_text (str): New text to display.
170 
171         Returns:
172             CustomEvent: Object containing the new text to alert state of a text
    update.
173         """
174         super().set_text(new_text)
175         return CustomEvent(**vars(self._event), text=self.text)
176 
177     def process_event(self, event):
178         """
179         Processes Pygame events.
180 
181         Args:
182             event (pygame.Event): Event to process.
183 
184         Returns:
185             CustomEvent: Object containing the new text to alert state of a text
    update.
186         """
187         previous_state = self.get_widget_state()
188         super().process_event(event)
189         current_state = self.get_widget_state()
190 
191         match event.type:
192             case pygame.MOUSEMOTION:
193                 if self._cursor_index is None:
194                     return
195 
196                 # If mouse is hovering over widget, turn mouse cursor into an I-
    beam
197                 if self.rect.collidepoint(event.pos):
198                     if cursor.get_mode() != CursorMode.IBEAM:
199                         cursor.set_mode(CursorMode.IBEAM)
200                 else:
201                     if cursor.get_mode() == CursorMode.IBEAM:
202                         cursor.set_mode(CursorMode.ARROW)
203 
204                 return
205 
206             case pygame.MOUSEBUTTONUP:
207                 # When user selects widget
208                 if previous_state == WidgetState.PRESS:
209                     self.focus_input(event.pos)
210                 # When user unselects widget
211                 if current_state == WidgetState.BASE and self._cursor_index is not
     None:
212                     self.unfocus_input()
```

```
213                     return CustomEvent(**vars(self._event), text=self.text)
214
215             case pygame.KEYDOWN:
216                 if self._cursor_index is None:
217                     return
218
219                 # Handling Ctrl-C and Ctrl-V shortcuts
220                 if event.mod & (pygame.KMOD_CTRL):
221                     if event.key == pygame.K_c:
222                         pyperclip.copy(self.text)
223                         logger.info(f'COPIED {self.text}')
224
225                     elif event.key == pygame.K_v:
226                         pasted_text = pyperclip.paste()
227                         pasted_text = ''.join(char for char in pasted_text if 32
    <= ord(char) <= 127)
228                         self._text = self._text[:self._cursor_index] + pasted_text
     + self._text[self._cursor_index:]
229                         self._cursor_index += len(pasted_text)
230
231                     elif event.key == pygame.K_BACKSPACE or event.key == pygame.
    K_DELETE:
232                         self._text = ''
233                         self._cursor_index = 0
234
235                     self.resize_text()
236                     self.set_image()
237                     self.set_geometry()
238
239                     return
240
241                 match event.key:
242                     case pygame.K_BACKSPACE:
243                         if self._cursor_index > 0:
244                             self._text = self._text[:self._cursor_index - 1] +
    self._text[self._cursor_index:]
245                         self._cursor_index = max(0, self._cursor_index - 1)
246
247                     case pygame.K_RIGHT:
248                         self._cursor_index = min(len(self._text), self.
    _cursor_index + 1)
249
250                     case pygame.K_LEFT:
251                         self._cursor_index = max(0, self._cursor_index - 1)
252
253                     case pygame.K_ESCAPE:
254                         self.unfocus_input()
255                         return CustomEvent(**vars(self._event), text=self.text)
256
257                     case pygame.K_RETURN:
258                         self.unfocus_input()
259                         return CustomEvent(**vars(self._event), text=self.text)
260
261                     case _:
262                         if not event.unicode:
263                             return
264
265                         potential_text = self._text[:self._cursor_index] + event.
    unicode + self._text[self._cursor_index:]
266
267                         # Validator lambda function used to check if inputted text
     is valid before displaying
```

```
268                         # e.g. Time control input has a validator function
    checking if text represents a float
269                         if self._validator(potential_text) is False:
270                             return
271 
272                         self._text = potential_text
273                         self._cursor_index += 1
274 
275                 self._blinking_cooldown += 1
276                 animation.set_timer(500, lambda: self.subtract_blinking_cooldown
    (1))
277 
278                 self.resize_text()
279                 self.set_image()
280                 self.set_geometry()
281 
282     def subtract_blinking_cooldown(self, cooldown):
283         """
284         Subtracts blinking cooldown after certain timeframe. When
    blinking_cooldown is 1, cursor is able to be drawn.
285 
286         Args:
287             cooldown (float): Duration before cursor can no longer be drawn.
288         """
289         self._blinking_cooldown = self._blinking_cooldown - cooldown
290 
291     def set_image(self):
292         """
293         Draws text input widget to image.
294         """
295         super().set_image()
296 
297         if self._cursor_index is not None:
298             scaled_cursor = pygame.transform.scale(self._empty_cursor, self.
    cursor_size)
299             scaled_cursor.fill(self._cursor_colour)
300             self.image.blit(scaled_cursor, self.cursor_position)
301 
302     def update(self):
303         """
304         Overrides based update method, to handle cursor blinking.
305         """
306         super().update()
307         # Calculate if cursor should be shown or not
308         cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
309         if cursor_frame == 1 and self._blinking_cooldown == 0:
310             self._cursor_colour = (0, 0, 0, 0)
311         else:
312             self._cursor_colour = self._cursor_colour_copy
313         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`

```
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__)

# TEMP
CPU_LIMIT_MS = 1500000

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)

```

```
62      def alert_listeners(self, event):
63          """
64          Alerts all registered classes of an event by calling their listener
    function.
65
66          Args:
67              event (GameEventType): Event to pass as argument.
68
69          Raises:
70              Exception: If an unrecgonised event tries to be passed onto listeners.
71          """
72          for parent_class, listeners in self._listeners.items():
73              match event.type:
74                  case GameEventType.UPDATE_PIECES:
75                      if parent_class in 'game':
76                          for listener in listeners: listener(event)
77
78                  case GameEventType.SET_LASER:
79                      if parent_class == 'game':
80                          for listener in listeners: listener(event)
81
82                  case GameEventType.PAUSE_CLICK:
83                      if parent_class in ['pause', 'game']:
84                          for listener in listeners:
85                              listener(event)
86
87                  case _:
88                      raise Exception('Unhandled event type (GameModel.
    alert_listeners)')
89
90      def set_winner(self, colour=None):
91          """
92          Sets winner.
93
94          Args:
95              colour (Colour, optional): Describes winnner colour, or draw. Defaults
     to None.
96          """
97          self.states['WINNER'] = colour
98
99      def toggle_paused(self):
100         """
101         Toggles pause screen, and alerts pause view.
102         """
103         self.states['PAUSED'] = not self.states['PAUSED']
104         game_event = CustomEvent.create_event(GameEventType.PAUSE_CLICK)
105         self.alert_listeners(game_event)
106
107     def get_terminal_move(self):
108         """
109         Debugging method for inputting a move from the terminal.
110
111         Returns:
112             Move: Parsed move.
113         """
114         while True:
115             try:
116                 move_type = ip_helpers.parse_move_type(input('Input move type (m/r
    ): '))
117                 src_square = ip_helpers.parse_notation(input("From: "))
118                 dest_square = ip_helpers.parse_notation(input("To: "))
119                 rotation = ip_helpers.parse_rotation(input("Enter rotation (a/b/c/
```

```
d): "))
120                 return Move.instance_from_notation(move_type, src_square,
    dest_square, rotation)
121             except ValueError as error:
122                 logger.warning('Input error (Board.get_move): ' + str(error))
123 
124     def make_move(self, move):
125         """
126         Takes a Move object and applies it to the board.
127 
128         Args:
129             move (Move): Move to apply.
130         """
131         colour = self._board.bitboards.get_colour_on(move.src)
132         piece = self._board.bitboards.get_piece_on(move.src, colour)
133         # Apply move and get results of laser trajectory
134         laser_result = self._board.apply_move(move, add_hash=True)
135 
136         self.alert_listeners(CustomEvent.create_event(GameEventType.SET_LASER,
    laser_result=laser_result))
137 
138         # Sets new active colour and checks for a win
139         self.states['ACTIVE_COLOUR'] = self._board.get_active_colour()
140         self.set_winner(self._board.check_win())
141 
142         move_notation = move.to_notation(colour, piece, laser_result.
    hit_square_bitboard)
143 
144         self.alert_listeners(CustomEvent.create_event(GameEventType.UPDATE_PIECES,
     move_notation=move_notation))
145 
146         # Adds move to move history list for review screen
147         self.states['MOVES'].append({
148             'time': {
149                 Colour.BLUE: GAME_WIDGETS['blue_timer'].get_time(),
150                 Colour.RED: GAME_WIDGETS['red_timer'].get_time()
151             },
152             'move': move_notation,
153             'laserResult': laser_result
154         })
155 
156     def make_cpu_move(self):
157         """
158         Starts CPU calculations on the separate thread.
159         """
160         self.states['AWAITING_CPU'] = True
161 
162         # Employ time management system to kill search if using an iterative
    deepening CPU
163         # if isinstance(self._cpu, IDMinimaxCPU):
164         #     move_id = getrandbits(32)
165         #     self._cpu_thread.start_cpu(self.get_board(), id=move_id)
166         #     animation.set_timer(CPU_LIMIT_MS, lambda: self._cpu_thread.stop_cpu(
    id=move_id))
167         # else:
168         self._cpu_thread.start_cpu(self.get_board())
169 
170     def cpu_callback(self, move):
171         """
172         Callback function passed to CPU thread. Called when CPU stops processing.
173 
174         Args:
```

```
175             move (Move): Move that CPU found.
176         """
177         if self.states['WINNER'] is None:
178             # CPU move passed back to main thread by reassigning variable
179             self._cpu_move = move
180             self.states['AWAITING_CPU'] = False
181 
182     def check_cpu(self):
183         """
184         Constantly checks if CPU calculations are finished, so that make_move can be run on the main thread.
185         """
186         if self._cpu_move is not None:
187             self.make_move(self._cpu_move)
188             self._cpu_move = None
189 
190     def kill_thread(self):
191         """
192         Interrupt and kill CPU thread.
193         """
194         self._cpu_thread.kill_thread()
195         self.states['AWAITING_CPU'] = False
196 
197     def is_selectable(self, bitboard):
198         """
199         Checks if square is occupied by a piece of the current active colour.
200 
201         Args:
202             bitboard (int): Bitboard representing single square.
203 
204         Returns:
205             bool: True if square is occupied by a piece of the current active colour. False if not.
206         """
207         return is_occupied(self._board.bitboards.combined_colour_bitboards[self.states['ACTIVE_COLOUR']], bitboard)
208 
209     def get_available_moves(self, bitboard):
210         """
211         Gets all surrounding empty squares. Used for drawing overlay.
212 
213         Args:
214             bitboard (int): Bitboard representing single center square.
215 
216         Returns:
217             int: Bitboard representing all empty surrounding squares.
218         """
219         if (bitboard & self._board.get_all_active_pieces()) != EMPTY_BB:
220             return self._board.get_valid_squares(bitboard)
221 
222         return EMPTY_BB
223 
224     def get_piece_list(self):
225         """
226         Returns:
227             list[Piece, ...]: Array of all pieces on the board.
228         """
229         return self._board.get_piece_list()
230 
231     def get_piece_info(self, bitboard):
232         """
233         Args:
```

```
234             bitboard (int): Square containing piece.
235
236         Returns:
237             tuple[Colour, Rotation, Piece]: Piece information.
238         """
239         colour = self._board.bitboards.get_colour_on(bitboard)
240         rotation = self._board.bitboards.get_rotation_on(bitboard)
241         piece = self._board.bitboards.get_piece_on(bitboard, colour)
242         return (piece, colour, rotation)
243
244     def get_fen_string(self):
245         return encode_fen_string(self._board.bitboards)
246
247     def get_board(self):
248         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 appropriate handling function within the view class is called by mapping the event type to the corresponding handler function.

game_view.py

```
1  import pygame
2  from data.utils.enums import Colour, StatusText, Miscellaneous, ShaderType
3  from data.states.game.components.overlay_draw import OverlayDraw
4  from data.states.game.components.capture_draw import CaptureDraw
5  from data.states.game.components.piece_group import PieceGroup
6  from data.states.game.components.laser_draw import LaserDraw
7  from data.states.game.components.father import DragAndDrop
8  from data.helpers.bitboard_helpers import bitboard_to_coords
9  from data.helpers.board_helpers import screen_pos_to_coords
10 from data.states.game.widget_dict import GAME_WIDGETS
11 from data.components.custom_event import CustomEvent
12 from data.components.widget_group import WidgetGroup
13 from data.utils.event_types import GameEventType
14 from data.managers.window import window
15 from data.managers.audio import audio
16 from data.utils.assets import SFX
17
18 class GameView:
19     def __init__(self, model):
20         self._model = model
21         self._hide_pieces = False
22         self._selected_coords = None
23         self._event_to_func_map = {
24             GameEventType.UPDATE_PIECES: self.handle_update_pieces,
25             GameEventType.SET_LASER: self.handle_set_laser,
26             GameEventType.PAUSE_CLICK: self.handle_pause,
27         }
28
29         # Register model event handling with process_model_event()
30         self._model.register_listener(self.process_model_event, 'game')
31
32         # Initialise WidgetGroup with map of widgets
33         self._widget_group = WidgetGroup(GAME_WIDGETS)
34         self._widget_group.handle_resize(window.size)
35         self.initialise_widgets()
```

```
36
37          self._laser_draw = LaserDraw(self.board_position, self.board_size)
38          self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
39          self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
40          self._capture_draw = CaptureDraw(self.board_position, self.board_size)
41          self._piece_group = PieceGroup()
42          self.handle_update_pieces()
43
44          self.set_status_text(StatusText.PLAYER_MOVE)
45
46      @property
47      def board_position(self):
48          return GAME_WIDGETS['chessboard'].position
49
50      @property
51      def board_size(self):
52          return GAME_WIDGETS['chessboard'].size
53
54      @property
55      def square_size(self):
56          return self.board_size[0] / 10
57
58      def initialise_widgets(self):
59          """
60          Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
61          """
62          GAME_WIDGETS['move_list'].reset_move_list()
63          GAME_WIDGETS['move_list'].kill()
64          GAME_WIDGETS['help'].kill()
65          GAME_WIDGETS['tutorial'].kill()
66
67          GAME_WIDGETS['scroll_area'].set_image()
68
69          GAME_WIDGETS['chessboard'].refresh_board()
70
71          GAME_WIDGETS['blue_piece_display'].reset_piece_list()
72          GAME_WIDGETS['red_piece_display'].reset_piece_list()
73
74      def set_status_text(self, status):
75          """
76          Sets text on status text widget.
77
78          Args:
79              status (StatusText): The game stage for which text should be displayed
     for.
80          """
81          match status:
82              case StatusText.PLAYER_MOVE:
83                  GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
    ACTIVE_COLOUR'].name}'s turn to move")
84              case StatusText.CPU_MOVE:
85                  GAME_WIDGETS['status_text'].set_text("CPU thinking...") # CPU
    calculating a crazy move...
86              case StatusText.WIN:
87                  if self._model.states['WINNER'] == Miscellaneous.DRAW:
88                      GAME_WIDGETS['status_text'].set_text("Game is a draw! Boring
    ...")
89                  else:
90                      GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
    WINNER'].name} won!")
91              case StatusText.DRAW:
92                  GAME_WIDGETS['status_text'].set_text("Game is a draw! Boring...")
```

```
93
94      def handle_resize(self):
95          """
96          Handles resizing of the window.
97          """
98          self._overlay_draw.handle_resize(self.board_position, self.board_size)
99          self._capture_draw.handle_resize(self.board_position, self.board_size)
100         self._piece_group.handle_resize(self.board_position, self.board_size)
101         self._laser_draw.handle_resize(self.board_position, self.board_size)
102         self._laser_draw.handle_resize(self.board_position, self.board_size)
103         self._widget_group.handle_resize(window.size)
104
105         if self._laser_draw.firing:
106             self.update_laser_mask()
107
108     def handle_update_pieces(self, event=None):
109         """
110         Callback function to update pieces after move.
111
112         Args:
113             event (GameEventType, optional): If updating pieces after player move,
     event contains move information. Defaults to None.
114             toggle_timers (bool, optional): Toggle timers on and off for new
    active colour. Defaults to True.
115         """
116         piece_list = self._model.get_piece_list()
117         self._piece_group.initialise_pieces(piece_list, self.board_position, self.
    board_size)
118
119         if event:
120             GAME_WIDGETS['move_list'].append_to_move_list(event.move_notation)
121             GAME_WIDGETS['scroll_area'].set_image()
122             audio.play_sfx(SFX['piece_move'])
123
124         # If active colour is starting colour, as player always moves first
125         if ['b', 'r'][self._model.states['ACTIVE_COLOUR']] == self._model.states['
    START_FEN_STRING'][-1]:
126             self.set_status_text(StatusText.PLAYER_MOVE)
127         else:
128             if self._model.states['CPU_ENABLED']:
129                 self.set_status_text(StatusText.CPU_MOVE)
130             else:
131                 self.set_status_text(StatusText.PLAYER_MOVE)
132
133         if self._model.states['TIME_ENABLED']:
134             self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
135             self.toggle_timer(self._model.states['ACTIVE_COLOUR'].
    get_flipped_colour(), False)
136
137         if self._model.states['WINNER'] is not None:
138             self.handle_game_end()
139
140         # Update occlusion mask for rays shader with new piece positions
141         self.update_laser_mask()
142
143     def handle_game_end(self, play_sfx=True):
144         self.toggle_timer(self._model.states['ACTIVE_COLOUR'], False)
145         self.toggle_timer(self._model.states['ACTIVE_COLOUR'].get_flipped_colour()
    , False)
146
147         if self._model.states['WINNER'] == Miscellaneous.DRAW:
148             self.set_status_text(StatusText.DRAW)
```

```
149             else:
150                 self.set_status_text(StatusText.WIN)
151 
152             if play_sfx:
153                 audio.play_sfx(SFX['sphinx_destroy_1'])
154                 audio.play_sfx(SFX['sphinx_destroy_2'])
155                 audio.play_sfx(SFX['sphinx_destroy_3'])
156 
157         def handle_set_laser(self, event):
158             """
159             Callback function to draw laser after move.
160 
161             Args:
162                 event (GameEventType): Contains laser trajectory information.
163             """
164             laser_result = event.laser_result
165 
166             # If laser has hit a piece
167             if laser_result.hit_square_bitboard:
168                 coords_to_remove = bitboard_to_coords(laser_result.hit_square_bitboard
    )
169                 self._piece_group.remove_piece(coords_to_remove)
170 
171                 if laser_result.piece_colour == Colour.BLUE:
172                     GAME_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit
    )
173                 elif laser_result.piece_colour == Colour.RED:
174                     GAME_WIDGETS['blue_piece_display'].add_piece(laser_result.
    piece_hit)
175 
176                 # Draw piece capture GFX
177                 self._capture_draw.add_capture(
178                     laser_result.piece_hit,
179                     laser_result.piece_colour,
180                     laser_result.piece_rotation,
181                     coords_to_remove,
182                     laser_result.laser_path[0][0],
183                     self._model.states['ACTIVE_COLOUR']
184                 )
185 
186             self._laser_draw.add_laser(laser_result, self._model.states['ACTIVE_COLOUR
    '])
187 
188         def handle_pause(self, event=None):
189             """
190             Callback function for pausing timer.
191 
192             Args:
193                 event (None): Event argument not used.
194             """
195             is_active = not(self._model.states['PAUSED'])
196             self.toggle_timer(self._model.states['ACTIVE_COLOUR'], is_active)
197 
198         def initialise_timers(self):
199             """
200             Initialises both timers with the correct amount of time and starts the
    timer for the active colour.
201             """
202             if self._model.states['TIME_ENABLED']:
203                 GAME_WIDGETS['blue_timer'].set_time(self._model.states['TIME'] * 60 *
    1000)
204                 GAME_WIDGETS['red_timer'].set_time(self._model.states['TIME'] * 60 *
```

```
        1000)
205         else:
206             GAME_WIDGETS['blue_timer'].kill()
207             GAME_WIDGETS['red_timer'].kill()
208 
209         self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
210 
211     def toggle_timer(self, colour, is_active):
212         """
213         Stops or resumes timer.
214 
215         Args:
216             colour (Colour): Timer to toggle.
217             is_active (bool): Whether to pause or resume timer.
218         """
219         if colour == Colour.BLUE:
220             GAME_WIDGETS['blue_timer'].set_active(is_active)
221         elif colour == Colour.RED:
222             GAME_WIDGETS['red_timer'].set_active(is_active)
223 
224     def update_laser_mask(self):
225         """
226         Uses pygame.mask to create a mask for the pieces.
227         Used for occluding the ray shader.
228         """
229         temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
230         self._piece_group.draw(temp_surface)
231         mask = pygame.mask.from_surface(temp_surface, threshold=127)
232         mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255,
        0, 0, 255))
233 
234         window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
235 
236     def draw(self):
237         """
238         Draws GUI and pieces onto the screen.
239         """
240         self._widget_group.update()
241         self._capture_draw.update()
242 
243         self._widget_group.draw()
244         self._overlay_draw.draw(window.screen)
245 
246         if self._hide_pieces is False:
247             self._piece_group.draw(window.screen)
248 
249         self._laser_draw.draw(window.screen)
250         self._drag_and_drop.draw(window.screen)
251         self._capture_draw.draw(window.screen)
252 
253     def process_model_event(self, event):
254         """
255         Registered listener function for handling GameModel events.
256         Each event is mapped to a callback function, and the appropiate one is run
        .
257 
258         Args:
259             event (GameEventType): Game event to process.
260 
261         Raises:
262             KeyError: If an unrecgonised event type is passed as the argument.
263         """
```

```
264         try:
265             self._event_to_func_map.get(event.type)(event)
266         except:
267             raise KeyError('Event type not recognized in Game View (GameView.
    process_model_event):', event.type)
268 
269     def set_overlay_coords(self, available_coords_list, selected_coord):
270         """
271         Set board coordinates for potential moves overlay.
272 
273         Args:
274             available_coords_list (list[tuple[int, int]], ...): Array of
    coordinates
275             selected_coord (list[int, int]): Coordinates of selected piece.
276         """
277         self._selected_coords = selected_coord
278         self._overlay_draw.set_selected_coords(selected_coord)
279         self._overlay_draw.set_available_coords(available_coords_list)
280 
281     def get_selected_coords(self):
282         return self._selected_coords
283 
284     def set_dragged_piece(self, piece, colour, rotation):
285         """
286         Passes information of the dragged piece to the dragging drawing class.
287 
288         Args:
289             piece (Piece): Piece type of dragged piece.
290             colour (Colour): Colour of dragged piece.
291             rotation (Rotation): Rotation of dragged piece.
292         """
293         self._drag_and_drop.set_dragged_piece(piece, colour, rotation)
294 
295     def remove_dragged_piece(self):
296         """
297         Stops drawing dragged piece when user lets go of piece.
298         """
299         self._drag_and_drop.remove_dragged_piece()
300 
301     def convert_mouse_pos(self, event):
302         """
303         Passes information of what mouse cursor is interacting with to a
    GameController object.
304 
305         Args:
306             event (pygame.Event): Mouse event to process.
307 
308         Returns:
309             CustomEvent | None: Contains information what mouse is doing.
310         """
311         clicked_coords = screen_pos_to_coords(event.pos, self.board_position, self
    .board_size)
312 
313         if event.type == pygame.MOUSEBUTTONDOWN:
314             if clicked_coords:
315                 return CustomEvent.create_event(GameEventType.BOARD_CLICK, coords=
    clicked_coords)
316 
317             else:
318                 return None
319 
320         elif event.type == pygame.MOUSEBUTTONUP:
```

```
321             if self._drag_and_drop.dragged_sprite:
322                 piece, colour, rotation = self._drag_and_drop.get_dragged_info()
323                 piece_dragged = self._drag_and_drop.remove_dragged_piece()
324                 return CustomEvent.create_event(GameEventType.PIECE_DROP, coords=
    clicked_coords, piece=piece, colour=colour, rotation=rotation, remove_overlay=
    piece_dragged)
325 
326     def add_help_screen(self):
327         """
328         Draw help overlay when player clicks on the help button.
329         """
330         self._widget_group.add(GAME_WIDGETS['help'])
331         self._widget_group.handle_resize(window.size)
332 
333     def add_tutorial_screen(self):
334         """
335         Draw tutorial overlay when player clicks on the tutorial button.
336         """
337         self._widget_group.add(GAME_WIDGETS['tutorial'])
338         self._widget_group.handle_resize(window.size)
339         self._hide_pieces = True
340 
341     def remove_help_screen(self):
342         GAME_WIDGETS['help'].kill()
343 
344     def remove_tutorial_screen(self):
345         GAME_WIDGETS['tutorial'].kill()
346         self._hide_pieces = False
347 
348     def process_widget_event(self, event):
349         """
350         Passes Pygame event to WidgetGroup to allow individual widgets to process
    events.
351 
352         Args:
353             event (pygame.Event): Event to process.
354 
355         Returns:
356             CustomEvent | None: A widget event.
357         """
358         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

```
1  import pygame
2  from data.helpers import bitboard_helpers as bb_helpers
3  from data.utils.enums import MoveType, Miscellaneous
4  from data.states.game.components.move import Move
5  from data.utils.event_types import GameEventType
6  from data.managers.logs import initialise_logger
7 
8  logger = initialise_logger(__name__)
9 
10 class GameController:
11     def __init__(self, model, view, win_view, pause_view, to_menu, to_review,
    to_new_game):
```

```
12          self._model = model
13          self._view = view
14          self._win_view = win_view
15          self._pause_view = pause_view
16
17          self._to_menu = to_menu
18          self._to_review = to_review
19          self._to_new_game = to_new_game
20
21          self._view.initialise_timers()
22          self._win_view.set_win_type('CAPTURE')
23
24      def cleanup(self, next):
25          """
26          Handles game quit, either leaving to main menu or restarting a new game.
27
28          Args:
29              next (str): New state to switch to.
30          """
31          self._model.kill_thread()
32
33          if next == 'menu':
34              self._to_menu()
35          elif next == 'game':
36              self._to_new_game()
37          elif next == 'review':
38              self._to_review()
39
40      def make_move(self, move):
41          """
42          Handles player move.
43
44          Args:
45              move (Move): Move to make.
46          """
47          self._model.make_move(move)
48          self._view.set_overlay_coords([], None)
49
50          if self._model.states['CPU_ENABLED']:
51              self._model.make_cpu_move()
52
53          if self._model.states['WINNER'] == Miscellaneous.DRAW:
54              self._win_view.set_win_type('DRAW')
55
56      def handle_pause_event(self, event):
57          """
58          Processes events when game is paused.
59
60          Args:
61              event (GameEventType): Event to process.
62
63          Raises:
64              Exception: If event type is unrecognised.
65          """
66          game_event = self._pause_view.convert_mouse_pos(event)
67
68          if game_event is None:
69              return
70
71          match game_event.type:
72              case GameEventType.PAUSE_CLICK:
73                  self._model.toggle_paused()
```

```
74
75              case GameEventType.MENU_CLICK:
76                  self.cleanup('menu')
77
78              case _:
79                  raise Exception('Unhandled event type (GameController.handle_event
    )')
80
81      def handle_winner_event(self, event):
82          """
83          Processes events when game is over.
84
85          Args:
86              event (GameEventType): Event to process.
87
88          Raises:
89              Exception: If event type is unrecognised.
90          """
91          game_event = self._win_view.convert_mouse_pos(event)
92
93          if game_event is None:
94              return
95
96          match game_event.type:
97              case GameEventType.MENU_CLICK:
98                  self.cleanup('menu')
99                  return
100
101             case GameEventType.GAME_CLICK:
102                 self.cleanup('game')
103                 return
104
105             case GameEventType.REVIEW_CLICK:
106                 self.cleanup('review')
107
108             case _:
109                 raise Exception('Unhandled event type (GameController.handle_event
    )')
110
111     def handle_game_widget_event(self, event):
112         """
113         Processes events for game GUI widgets.
114
115         Args:
116             event (GameEventType): Event to process.
117
118         Raises:
119             Exception: If event type is unrecognised.
120
121         Returns:
122             CustomEvent | None: A widget event.
123         """
124         widget_event = self._view.process_widget_event(event)
125
126         if widget_event is None:
127             return None
128
129         match widget_event.type:
130             case GameEventType.ROTATE_PIECE:
131                 src_coords = self._view.get_selected_coords()
132
133                 if src_coords is None:
```

```
134                     logger.info('None square selected')
135                     return
136
137                 move = Move.instance_from_coords(MoveType.ROTATE, src_coords,
    src_coords, rotation_direction=widget_event.rotation_direction)
138                 self.make_move(move)
139
140             case GameEventType.RESIGN_CLICK:
141                 self._model.set_winner(self._model.states['ACTIVE_COLOUR'].
    get_flipped_colour())
142                 self._view.handle_game_end(play_sfx=False)
143                 self._win_view.set_win_type('RESIGN')
144
145             case GameEventType.DRAW_CLICK:
146                 self._model.set_winner(Miscellaneous.DRAW)
147                 self._view.handle_game_end(play_sfx=False)
148                 self._win_view.set_win_type('DRAW')
149
150             case GameEventType.TIMER_END:
151                 if self._model.states['TIME_ENABLED']:
152                     self._model.set_winner(widget_event.active_colour.
    get_flipped_colour())
153                     self._win_view.set_win_type('TIME')
154                     self._view.handle_game_end(play_sfx=False)
155
156             case GameEventType.MENU_CLICK:
157                 self.cleanup('menu')
158
159             case GameEventType.HELP_CLICK:
160                 self._view.add_help_screen()
161
162             case GameEventType.TUTORIAL_CLICK:
163                 self._view.add_tutorial_screen()
164
165             case _:
166                 raise Exception('Unhandled event type (GameController.handle_event
    )')
167
168         return widget_event.type
169
170     def check_cpu(self):
171         """
172         Checks if CPU calculations are finished every frame.
173         """
174         if self._model.states['CPU_ENABLED'] and self._model.states['AWAITING_CPU'
    ] is False:
175             self._model.check_cpu()
176
177     def handle_game_event(self, event):
178         """
179         Processes Pygame events for main game.
180
181         Args:
182             event (pygame.Event): If event type is unrecognised.
183
184         Raises:
185             Exception: If event type is unrecognised.
186         """
187         # Pass event for widgets to process
188         widget_event = self.handle_game_widget_event(event)
189
190         if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
```

```
KEYDOWN]:
191             if event.type != pygame.KEYDOWN:
192                 game_event = self._view.convert_mouse_pos(event)
193             else:
194                 game_event = None
195 
196             if game_event is None:
197                 if widget_event is None:
198                     if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
199                         # If user releases mouse click not on a widget
200                         self._view.remove_help_screen()
201                         self._view.remove_tutorial_screen()
202                     if event.type == pygame.MOUSEBUTTONUP:
203                         # If user releases mouse click on neither a widget or
    board
204                         self._view.set_overlay_coords(None, None)
205 
206                 return
207 
208             match game_event.type:
209                 case GameEventType.BOARD_CLICK:
210                     if self._model.states['AWAITING_CPU']:
211                         return
212 
213                     clicked_coords = game_event.coords
214                     clicked_bitboard = bb_helpers.coords_to_bitboard(
    clicked_coords)
215                     selected_coords = self._view.get_selected_coords()
216 
217                     if selected_coords:
218                         if clicked_coords == selected_coords:
219                             # If clicking on an already selected square, start
    dragging piece on that square
220                             self._view.set_dragged_piece(*self._model.
    get_piece_info(clicked_bitboard))
221                             return
222 
223                         selected_bitboard = bb_helpers.coords_to_bitboard(
    selected_coords)
224                         available_bitboard = self._model.get_available_moves(
    selected_bitboard)
225 
226                         if bb_helpers.is_occupied(clicked_bitboard,
    available_bitboard):
227                             # If the newly clicked square is not the same as the
    old one, and is an empty surrounding square, make a move
228                             move = Move.instance_from_coords(MoveType.MOVE,
    selected_coords, clicked_coords)
229                             self.make_move(move)
230                         else:
231                             # If the newly clicked square is not the same as the
    old one, but is an invalid square, unselect the currently selected square
232                             self._view.set_overlay_coords(None, None)
233 
234                     # Select hovered square if it is same as active colour
235                     elif self._model.is_selectable(clicked_bitboard):
236                         available_bitboard = self._model.get_available_moves(
    clicked_bitboard)
237                         self._view.set_overlay_coords(bb_helpers.
    bitboard_to_coords_list(available_bitboard), clicked_coords)
238                         self._view.set_dragged_piece(*self._model.get_piece_info(
    clicked_bitboard))
```

```
239
240                 case GameEventType.PIECE_DROP:
241                     hovered_coords = game_event.coords
242
243                     # if piece is dropped onto the board
244                     if hovered_coords:
245                         hovered_bitboard = bb_helpers.coords_to_bitboard(
    hovered_coords)
246                         selected_coords = self._view.get_selected_coords()
247                         selected_bitboard = bb_helpers.coords_to_bitboard(
    selected_coords)
248                         available_bitboard = self._model.get_available_moves(
    selected_bitboard)
249
250                         if bb_helpers.is_occupied(hovered_bitboard,
    available_bitboard):
251                             # Make a move if mouse is hovered over an empty
    surrounding square
252                             move = Move.instance_from_coords(MoveType.MOVE,
    selected_coords, hovered_coords)
253                             self.make_move(move)
254
255                     if game_event.remove_overlay:
256                         self._view.set_overlay_coords(None, None)
257
258                     self._view.remove_dragged_piece()
259
260                 case _:
261                     raise Exception('Unhandled event type (GameController.
    handle_event)', game_event.type)
262
263     def handle_event(self, event):
264         """
265         Passes a Pygame event to the correct handling function according to the
    game state.
266
267         Args:
268             event (pygame.Event): Event to process.
269         """
270         if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
    MOUSEMOTION, pygame.KEYDOWN]:
271             if self._model.states['PAUSED']:
272                 self.handle_pause_event(event)
273             elif self._model.states['WINNER'] is not None:
274                 self.handle_winner_event(event)
275             else:
276                 self.handle_game_event(event)
277
278         if event.type == pygame.KEYDOWN:
279             if event.key == pygame.K_ESCAPE:
280                 self._model.toggle_paused()
281             # Debug shortcut to kill CPU
282             elif event.key == pygame.K_l:
283                 logger.info('\nSTOPPING CPU')
284                 self._model._cpu_thread.stop_cpu()
```

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):
        """
```

```
57          Gets the active colour.
58
59          Returns:
60              Colour: The active colour.
61          """
62          return self.bitboards.active_colour
63
64      def to_hash(self):
65          """
66          Gets the hash of the current board state.
67
68          Returns:
69              int: A Zobrist hash.
70          """
71          return self.bitboards.get_hash()
72
73      def check_win(self):
74          """
75          Checks for a Pharaoh capture or threefold-repetition.
76
77          Returns:
78              Colour | Miscellaneous: The winning colour, or Miscellaneous.DRAW.
79          """
80          for colour in Colour:
81              if self.bitboards.get_piece_bitboard(Piece.PHARAOH, colour) ==
    EMPTY_BB:
82                  return colour.get_flipped_colour()
83
84          if self.hash_list.count(self.hash_list[-1]) >= 3:
85              return Miscellaneous.DRAW
86
87          return None
88
89      def apply_move(self, move, fire_laser=True, add_hash=False):
90          """
91          Applies a move to the board.
92
93          Args:
94              move (Move): The move to apply.
95              fire_laser (bool): Whether to fire the laser after the move.
96              add_hash (bool): Whether to add the board state hash to the hash list.
97
98          Returns:
99              Laser: The laser trajectory result.
100         """
101         piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
    active_colour)
102
103         if piece_symbol is None:
104             raise ValueError(f'Invalid move - no piece found on source square. {
    move}')
105         elif piece_symbol == Piece.SPHINX:
106             raise ValueError(f'Invalid move - sphinx piece is immovable. {move}')
107
108         if move.move_type == MoveType.MOVE:
109             possible_moves = self.get_valid_squares(move.src)
110             if bb_helpers.is_occupied(move.dest, possible_moves) is False:
111                 raise ValueError('Invalid move - destination square is occupied')
112
113             piece_rotation = self.bitboards.get_rotation_on(move.src)
114
115             self.bitboards.update_move(move.src, move.dest)
```

```
116             self.bitboards.update_rotation(move.src, move.dest, piece_rotation)
117
118         elif move.move_type == MoveType.ROTATE:
119             piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
    active_colour)
120             piece_rotation = self.bitboards.get_rotation_on(move.src)
121
122             if move.rotation_direction == RotationDirection.CLOCKWISE:
123                 new_rotation = piece_rotation.get_clockwise()
124             elif move.rotation_direction == RotationDirection.ANTICLOCKWISE:
125                 new_rotation = piece_rotation.get_anticlockwise()
126
127             self.bitboards.update_rotation(move.src, move.src, new_rotation)
128
129         laser = None
130         if fire_laser:
131             laser = self.fire_laser(add_hash)
132
133         if add_hash:
134             self.hash_list.append(self.bitboards.get_hash())
135
136         self.bitboards.flip_colour()
137
138         return laser
139
140     def undo_move(self, move, laser_result):
141         """
142         Undoes a move on the board.
143
144         Args:
145             move (Move): The move to undo.
146             laser_result (Laser): The laser trajectory result.
147         """
148         self.bitboards.flip_colour()
149
150         if laser_result.hit_square_bitboard:
151             # Get info of destroyed piece, and add it to the board again
152             src = laser_result.hit_square_bitboard
153             piece = laser_result.piece_hit
154             colour = laser_result.piece_colour
155             rotation = laser_result.piece_rotation
156
157             self.bitboards.set_square(src, piece, colour)
158             self.bitboards.clear_rotation(src)
159             self.bitboards.set_rotation(src, rotation)
160
161         # Create new Move object that is the inverse of the passed move
162         if move.move_type == MoveType.MOVE:
163             reversed_move = Move.instance_from_bitboards(MoveType.MOVE, move.dest,
     move.src)
164         elif move.move_type == MoveType.ROTATE:
165             reversed_move = Move.instance_from_bitboards(MoveType.ROTATE, move.src
    , move.src, move.rotation_direction.get_opposite())
166
167         self.apply_move(reversed_move, fire_laser=False)
168         self.bitboards.flip_colour()
169
170     def remove_piece(self, square_bitboard):
171         """
172         Removes a piece from a given square.
173
174         Args:
```

```
175             square_bitboard (int): The bitboard representation of the square.
176         """
177         self.bitboards.clear_square(square_bitboard, Colour.BLUE)
178         self.bitboards.clear_square(square_bitboard, Colour.RED)
179         self.bitboards.clear_rotation(square_bitboard)
180 
181     def get_valid_squares(self, src_bitboard, colour=None):
182         """
183         Gets valid squares for a piece to move to.
184 
185         Args:
186             src_bitboard (int): The bitboard representation of the source square.
187             colour (Colour, optional): The active colour of the piece.
188 
189         Returns:
190             int: The bitboard representation of valid squares.
191         """
192         target_top_left = (src_bitboard & A_FILE_MASK & EIGHT_RANK_MASK) << 9
193         target_top_middle = (src_bitboard & EIGHT_RANK_MASK) << 10
194         target_top_right = (src_bitboard & J_FILE_MASK & EIGHT_RANK_MASK) << 11
195         target_middle_right = (src_bitboard & J_FILE_MASK) << 1
196 
197         target_bottom_right = (src_bitboard & J_FILE_MASK & ONE_RANK_MASK) >> 9
198         target_bottom_middle = (src_bitboard & ONE_RANK_MASK) >> 10
199         target_bottom_left = (src_bitboard & A_FILE_MASK & ONE_RANK_MASK)>> 11
200         target_middle_left = (src_bitboard & A_FILE_MASK) >> 1
201 
202         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
203 
204         if colour is not None:
205             valid_possible_moves = possible_moves & ~self.bitboards.
    combined_colour_bitboards[colour]
206         else:
207             valid_possible_moves = possible_moves & ~self.bitboards.
    combined_all_bitboard
208 
209         return valid_possible_moves
210 
211     def get_mobility(self, colour):
212         """
213         Gets all valid squares for a given colour.
214 
215         Args:
216             colour (Colour): The colour of the pieces.
217 
218         Returns:
219             int: The bitboard representation of all valid squares.
220         """
221         active_pieces = self.get_all_active_pieces(colour)
222         possible_moves = 0
223 
224         for square in bb_helpers.occupied_squares(active_pieces):
225             possible_moves += bb_helpers.pop_count(self.get_valid_squares(square))
226 
227         return possible_moves
228 
229     def get_all_active_pieces(self, colour=None):
230         """
231         Gets all active pieces for the current player.
232 
```

```
233         Args:
234             colour (Colour): Active colour of pieces to retrieve. Defaults to None
    .
235 
236         Returns:
237             int: The bitboard representation of all active pieces.
238         """
239         if colour is None:
240             colour = self.bitboards.active_colour
241 
242         active_pieces = self.bitboards.combined_colour_bitboards[colour]
243         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
244         return active_pieces ^ sphinx_bitboard
245 
246     def fire_laser(self, remove_hash):
247         """
248         Fires the laser and removes hit pieces.
249 
250         Args:
251             remove_hash (bool): Whether to clear the hash list if a piece is hit.
252 
253         Returns:
254             Laser: The result of firing the laser.
255         """
256         laser = Laser(self.bitboards)
257 
258         if laser.hit_square_bitboard:
259             self.remove_piece(laser.hit_square_bitboard)
260 
261             if remove_hash:
262                 self.hash_list = [] # Remove all hashes for threefold repetition,
    as the position is impossible to be repeated after a piece is removed
263         return laser
264 
265     def generate_square_moves(self, src):
266         """
267         Generates all valid moves for a piece on a given square.
268 
269         Args:
270             src (int): The bitboard representation of the source square.
271 
272         Yields:
273             Move: A valid move for the piece.
274         """
275         for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
276             yield Move(MoveType.MOVE, src, dest)
277 
278     def generate_all_moves(self, colour):
279         """
280         Generates all valid moves for a given colour.
281 
282         Args:
283             colour (Colour): The colour of the pieces.
284 
285         Yields:
286             Move: A valid move for the active colour.
287         """
288         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
289         # Remove source squares for Sphinx pieces, as they cannot be moved
290         sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour]
    ^ sphinx_bitboard
291 
```

```
292         for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
293             # Generate movement moves
294             yield from self.generate_square_moves(square)
295 
296             # Generate rotational moves
297             for rotation_direction in RotationDirection:
298                 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`

```
1  from data.utils.enums import Rank, File, Piece, Colour, Rotation, RotationIndex
2  from data.states.game.components.fen_parser import parse_fen_string
3  from data.states.game.cpu.zobrist_hasher import ZobristHasher
4  from data.helpers import bitboard_helpers as bb_helpers
5  from data.managers.logs import initialise_logger
6  from data.utils.constants import EMPTY_BB
7 
8  logger = initialise_logger(__name__)
9 
10 class BitboardCollection:
11     def __init__(self, fen_string):
12         self.piece_bitboards = [{char: EMPTY_BB for char in Piece}, {char:
    EMPTY_BB for char in Piece}]
13         self.combined_colour_bitboards = [EMPTY_BB, EMPTY_BB]
14         self.combined_all_bitboard = EMPTY_BB
15         self.rotation_bitboards = [EMPTY_BB, EMPTY_BB]
16         self.active_colour = Colour.BLUE
17         self._hasher = ZobristHasher()
18 
19         try:
20             if fen_string:
21                 self.piece_bitboards, self.combined_colour_bitboards, self.
    combined_all_bitboard, self.rotation_bitboards, self.active_colour =
    parse_fen_string(fen_string)
22                 self.initialise_hash()
23         except ValueError as error:
24             logger.error('Please input a valid FEN string:', error)
25             raise error
26 
27     def __str__(self):
28         """
29         Returns a string representation of the bitboards.
30 
31         Returns:
32             str: Bitboards formatted with piece type and colour shown.
33         """
34         characters = ''
35         for rank in reversed(Rank):
36             for file in File:
37                 bitboard = 1 << (rank * 10 + file)
38 
39                 colour = self.get_colour_on(bitboard)
40                 piece = self.get_piece_on(bitboard, Colour.BLUE) or self.
    get_piece_on(bitboard, Colour.RED)
41 
```

```
                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:
```

```
100             self._hasher.apply_red_move_hash()
101
102     def update_move(self, src, dest):
103         """
104         Updates the bitboards for a move.
105
106         Args:
107             src (int): The bitboard representation of the source square.
108             dest (int): The bitboard representation of the destination square.
109         """
110         piece = self.get_piece_on(src, self.active_colour)
111
112         self.clear_square(src, Colour.BLUE)
113         self.clear_square(dest, Colour.BLUE)
114         self.clear_square(src, Colour.RED)
115         self.clear_square(dest, Colour.RED)
116
117         self.set_square(dest, piece, self.active_colour)
118
119     def update_rotation(self, src, dest, new_rotation):
120         """
121         Updates the rotation bitboards for a move.
122
123         Args:
124             src (int): The bitboard representation of the source square.
125             dest (int): The bitboard representation of the destination square.
126             new_rotation (Rotation): The new rotation.
127         """
128         self.clear_rotation(src)
129         self.set_rotation(dest, new_rotation)
130
131     def clear_rotation(self, bitboard):
132         """
133         Clears the rotation for a given square.
134
135         Args:
136             bitboard (int): The bitboard representation of the square.
137         """
138         old_rotation = self.get_rotation_on(bitboard)
139         rotation_1, rotation_2 = self.rotation_bitboards
140         self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.clear_square(
    rotation_1, bitboard)
141         self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.clear_square
    (rotation_2, bitboard)
142
143         self._hasher.apply_rotation_hash(bitboard, old_rotation)
144
145     def clear_square(self, bitboard, colour):
146         """
147         Clears a square piece and rotation for a given colour.
148
149         Args:
150             bitboard (int): The bitboard representation of the square.
151             colour (Colour): The colour to clear.
152         """
153         piece = self.get_piece_on(bitboard, colour)
154
155         if piece is None:
156             return
157
158         piece_bitboard = self.get_piece_bitboard(piece, colour)
159         colour_bitboard = self.combined_colour_bitboards[colour]
```

```
160         all_bitboard = self.combined_all_bitboard
161 
162         self.piece_bitboards[colour][piece] = bb_helpers.clear_square(
    piece_bitboard, bitboard)
163         self.combined_colour_bitboards[colour] = bb_helpers.clear_square(
    colour_bitboard, bitboard)
164         self.combined_all_bitboard = bb_helpers.clear_square(all_bitboard,
    bitboard)
165 
166         self._hasher.apply_piece_hash(bitboard, piece, colour)
167 
168     def set_rotation(self, bitboard, rotation):
169         """
170         Sets the rotation for a given square.
171 
172         Args:
173             bitboard (int): The bitboard representation of the square.
174             rotation (Rotation): The rotation to set.
175         """
176         rotation_1, rotation_2 = self.rotation_bitboards
177         self._hasher.apply_rotation_hash(bitboard, rotation)
178 
179         match rotation:
180             case Rotation.UP:
181                 return
182             case Rotation.RIGHT:
183                 self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
    set_square(rotation_1, bitboard)
184                 return
185             case Rotation.DOWN:
186                 self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.
    set_square(rotation_2, bitboard)
187                 return
188             case Rotation.LEFT:
189                 self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
    set_square(rotation_1, bitboard)
190                 self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.
    set_square(rotation_2, bitboard)
191                 return
192             case _:
193                 raise ValueError('Invalid rotation input (bitboard.py):', rotation
    )
194 
195     def set_square(self, bitboard, piece, colour):
196         """
197         Sets a piece on a given square.
198 
199         Args:
200             bitboard (int): The bitboard representation of the square.
201             piece (Piece): The piece to set.
202             colour (Colour): The colour of the piece.
203         """
204         piece_bitboard = self.get_piece_bitboard(piece, colour)
205         colour_bitboard = self.combined_colour_bitboards[colour]
206         all_bitboard = self.combined_all_bitboard
207 
208         self.piece_bitboards[colour][piece] = bb_helpers.set_square(piece_bitboard
    , bitboard)
209         self.combined_colour_bitboards[colour] = bb_helpers.set_square(
    colour_bitboard, bitboard)
210         self.combined_all_bitboard = bb_helpers.set_square(all_bitboard, bitboard)
211 
```

```
212         self._hasher.apply_piece_hash(bitboard, piece, colour)
213 
214     def get_piece_bitboard(self, piece, colour):
215         """
216         Gets the bitboard for a piece type for a given colour.
217 
218         Args:
219             piece (Piece): The piece bitboard to get.
220             colour (Colour): The colour of the piece.
221 
222         Returns:
223             int: The bitboard representation for all squares occupied by that
    piece type.
224         """
225         return self.piece_bitboards[colour][piece]
226 
227     def get_piece_on(self, target_bitboard, colour):
228         """
229         Gets the piece on a given square for a given colour.
230 
231         Args:
232             target_bitboard (int): The bitboard representation of the square.
233             colour (Colour): The colour of the piece.
234 
235         Returns:
236             Piece: The piece on the square, or None if square is empty.
237         """
238         if not (bb_helpers.is_occupied(self.combined_colour_bitboards[colour],
    target_bitboard)):
239             return None
240 
241         return next(
242             (piece for piece in Piece if
243                 bb_helpers.is_occupied(self.get_piece_bitboard(piece, colour),
    target_bitboard)),
244             None)
245 
246     def get_rotation_on(self, target_bitboard):
247         """
248         Gets the rotation on a given square.
249 
250         Args:
251             target_bitboard (int): The bitboard representation of the square.
252 
253         Returns:
254             Rotation: The rotation on the square.
255         """
256         rotationBits = [bb_helpers.is_occupied(self.rotation_bitboards[
    RotationIndex.SECONDBIT], target_bitboard), bb_helpers.is_occupied(self.
    rotation_bitboards[RotationIndex.FIRSTBIT], target_bitboard)]
257 
258         match rotationBits:
259             case [False, False]:
260                 return Rotation.UP
261             case [False, True]:
262                 return Rotation.RIGHT
263             case [True, False]:
264                 return Rotation.DOWN
265             case [True, True]:
266                 return Rotation.LEFT
267 
268     def get_colour_on(self, target_bitboard):
```

```
269         """
270         Gets the colour of the piece on a given square.
271 
272         Args:
273             target_bitboard (int): The bitboard representation of the square.
274 
275         Returns:
276             Colour: The colour of the piece on the square.
277         """
278         for piece in Piece:
279             if self.get_piece_bitboard(piece, Colour.BLUE) & target_bitboard !=
    EMPTY_BB:
280                 return Colour.BLUE
281             elif self.get_piece_bitboard(piece, Colour.RED) & target_bitboard !=
    EMPTY_BB:
282                 return Colour.RED
283 
284     def get_piece_count(self, piece, colour):
285         """
286         Gets the count of a given piece type and colour.
287 
288         Args:
289             piece (Piece): The piece to count.
290             colour (Colour): The colour of the piece.
291 
292         Returns:
293             int: The number of that piece of that colour on the board.
294         """
295         return bb_helpers.pop_count(self.get_piece_bitboard(piece, colour))
296 
297     def get_hash(self):
298         """
299         Gets the Zobrist hash of the current board state.
300 
301         Returns:
302             int: The Zobrist hash.
303         """
304         return self._hasher.hash
305 
306     def convert_to_piece_list(self):
307         """
308         Converts all bitboards to a list of pieces.
309 
310         Returns:
311             list: Board represented as a 2D list of Piece and Rotation objects.
312         """
313         piece_list = []
314 
315         for i in range(80):
316             if x := self.get_piece_on(1 << i, Colour.BLUE):
317                 rotation = self.get_rotation_on(1 << i)
318                 piece_list.append((x.upper(), rotation))
319             elif y := self.get_piece_on(1 << i, Colour.RED):
320                 rotation = self.get_rotation_on(1 << i)
321                 piece_list.append((y, rotation))
322             else:
323                 piece_list.append(None)
324 
325         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

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
```

```
39
40          best_move = None
41
42          # Blue is the maximising player
43          if board.get_active_colour() == Colour.BLUE:
44              max_score = -Score.INFINITE
45
46              for move in board.generate_all_moves(Colour.BLUE):
47                  laser_result = board.apply_move(move)
48
49
50                  new_score = self.search(board, depth - 1, stop_event)[0]
51
52                  # if depth < self._max_depth:
53                  #     print('DEPTH', depth, new_score, move)
54
55                  if new_score > max_score:
56                      max_score = new_score
57                      best_move = move
58
59                      if new_score == (Score.CHECKMATE + self._max_depth):
60                          board.undo_move(move, laser_result)
61                          return max_score, best_move
62
63                  elif new_score == max_score:
64                      # If evaluated scores are equal, pick a random move
65                      best_move = choice([best_move, move])
66
67                  board.undo_move(move, laser_result)
68
69              return max_score, best_move
70
71          else:
72              min_score = Score.INFINITE
73
74              for move in board.generate_all_moves(Colour.RED):
75                  laser_result = board.apply_move(move)
76                  # print('DEPTH', depth, move)
77                  new_score = self.search(board, depth - 1, stop_event)[0]
78
79                  if new_score < min_score:
80                      # print('setting new', new_score, move)
81                      min_score = new_score
82                      best_move = move
83
84                      if new_score == (-Score.CHECKMATE - self._max_depth):
85                          board.undo_move(move, laser_result)
86                          return min_score, best_move
87
88                  elif new_score == min_score:
89                      best_move = choice([best_move, move])
90
91                  board.undo_move(move, laser_result)
92
93              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
```

```
55          if board.get_active_colour() == Colour.BLUE:
56              max_score = -Score.INFINITE
57
58              for move in self._orderer.get_moves(board, hint=hint, laser_coords=
    laser_coords):
59                  laser_result = board.apply_move(move)
60                  new_score = self.search(board, depth - 1, alpha, beta, stop_event,
     laser_coords=laser_result.pieces_on_trajectory)[0]
61
62                  if new_score > max_score:
63                      max_score = new_score
64                      best_move = move
65
66                  board.undo_move(move, laser_result)
67
68                  alpha = max(alpha, max_score)
69
70                  if beta <= alpha:
71                      self._stats['alpha_prunes'] += 1
72                      break
73
74              return max_score, best_move
75
76          else:
77              min_score = Score.INFINITE
78
79              for move in self._orderer.get_moves(board, hint=hint, laser_coords=
    laser_coords):
80                  laser_result = board.apply_move(move)
81                  new_score = self.search(board, depth - 1, alpha, beta, stop_event,
     laser_coords=laser_result.pieces_on_trajectory)[0]
82
83                  if new_score < min_score:
84                      min_score = new_score
85                      best_move = move
86
87                  board.undo_move(move, laser_result)
88
89                  beta = min(beta, min_score)
90                  if beta <= alpha:
91                      self._stats['beta_prunes'] += 1
92                      break
93
94              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

```
1 from data.states.game.cpu.transposition_table import TranspositionTable
2 from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU
3
4 class TranspositionTableMixin:
5     def __init__(self, *args, **kwargs):
6         super().__init__(*args, **kwargs)
7         self._table = TranspositionTable()
```

```
8
9       def find_move(self, *args, **kwargs):
10          self._table = TranspositionTable()
11          super().find_move(*args, **kwargs)
12
13      def search(self, board, depth, alpha, beta, stop_event, hint=None,
        laser_coords=None):
14          """
15          Searches transposition table for a cached move before running a full
        search if necessary.
16          Caches the searched result.
17
18          Args:
19              board (Board): The current board state.
20              depth (int): The current search depth.
21              alpha (int): The upper bound value.
22              beta (int): The lower bound value.
23              stop_event (threading.Event): Event used to kill search from an
        external class.
24
25          Returns:
26              tuple[int, Move]: The best score and the best move found.
27          """
28          hash = board.to_hash()
29          score, move = self._table.get_entry(hash, depth, alpha, beta)
30
31          if score is not None:
32              self._stats['cache_hits'] += 1
33              self._stats['nodes'] += 1
34
35              return score, move
36          else:
37              # If board hash entry not found in cache, run a full search
38              score, move = super().search(board, depth, alpha, beta, stop_event,
        hint)
39              self._table.insert_entry(score, move, hash, depth, alpha, beta)
40
41              return score, move
42
43  class TTMinimaxCPU(TranspositionTableMixin, ABMinimaxCPU):
44      def initialise_stats(self):
45          """
46          Initialises cache statistics to be logged.
47          """
48          super().initialise_stats()
49          self._stats['cache_hits'] = 0
50
51      def print_stats(self, score, move):
52          """
53          Logs the statistics for the search.
54
55          Args:
56              score (int): The best score found.
57              move (Move): The best move found.
58          """
59          # Calculate number of cached entries retrieved as a percentage of all
        nodes
60          self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] /
        self._stats['nodes'], 3)
61          self._stats['cache_entries'] = len(self._table._table)
62          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
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_pharaoh_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_pharaoh_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)}\n')

        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.
```

```
102
103         Returns:
104             int: Score on numerical representation of mobility.
105         """
106         number_of_moves = board.get_mobility(colour)
107         return number_of_moves * Score.MOVE
108
109     def evaluate_pharaoh_safety(self, board, colour):
110         """
111         Evaluates the safety of the Pharaoh for a given colour.
112
113         Args:
114             board (Board): The current board state.
115             colour (Colour): The colour to evaluate.
116
117         Returns:
118             int: Score representing mobility of the Pharaoh.
119         """
120         pharaoh_bitboard = board.bitboards.get_piece_bitboard(Piece.PHARAOH,
    colour)
121
122         if pharaoh_bitboard:
123             pharaoh_available_moves = pop_count(board.get_valid_squares(
    pharaoh_bitboard, colour))
124             return (8 - pharaoh_available_moves) * Score.PHARAOH_SAFETY
125         else:
126             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

```
1  import threading
2  import time
3  from data.managers.logs import initialise_logger
4
5  logger = initialise_logger(__name__)
6
7  class CPUThread(threading.Thread):
8      def __init__(self, cpu, verbose=False):
9          super().__init__()
10         self._stop_event = threading.Event()
11         self._running = True
12         self._verbose = verbose
13         self.daemon = True
14
15         self._board = None
16         self._cpu = cpu
17         self._id = None
18
19     def kill_thread(self):
20         """
21         Kills the CPU and terminates the thread by stopping the run loop.
22         """
23         self.stop_cpu(force=True)
```

```
24          self._running = False
25
26      def stop_cpu(self, id=None, force=False):
27          """
28          Kills the CPU's move search.
29
30          Args:
31              id (int, optional): Id of search to kill, only kills if matching.
32              force (bool, optional): Forcibly kill search regardless of id.
33          """
34          if self._id == id or force:
35              self._stop_event.set()
36              self._board = None
37
38      def start_cpu(self, board, id=None):
39          """
40          Starts the CPU's move search.
41
42          Args:
43              board (Board): The current board state.
44              id (int, optional): Id of current search.
45          """
46          self._stop_event.clear()
47          self._board = board
48          self._id = id
49
50      def run(self):
51          """
52          Periodically checks if the board variable is set.
53          If it is, then starts CPU search.
54          """
55          while self._running:
56              if self._board and self._cpu:
57                  self._cpu.find_move(self._board, self._stop_event)
58                  self.stop_cpu()
59              else:
60                  time.sleep(1)
61                  if self._verbose:
62                      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

```
1  from random import randint
2  from data.helpers.bitboard_helpers import bitboard_to_index
3  from data.utils.enums import Piece, Colour, Rotation
4
5  # Initialise random values for each piece type on every square
6  # (5 x 2 colours) pieces + 4 rotations, for 80 squares
7  zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)]
8  # Hash for when the red player's move
9  red_move_hash = randint(0, 2 ** 64)
10
11 # Maps piece to the correct random value
12 piece_lookup = {
13     Colour.BLUE: {
```

```
        piece: i for i, piece in enumerate(Piece)
    },
    Colour.RED: {
        piece: i + 5 for i, piece in enumerate(Piece)
    },
}

# Maps rotation to the correct random value
rotation_lookup = {
    rotation: i + 10 for i, rotation in enumerate(Rotation)
}

class ZobristHasher:
    def __init__(self):
        self.hash = 0

    def get_piece_hash(self, index, piece, colour):
        """
        Gets the random value for the piece type on the given square.

        Args:
            index (int): The index of the square.
            piece (Piece): The piece on the square.
            colour (Colour): The colour of the piece.

        Returns:
            int: A 64-bit value.
        """
        piece_index = piece_lookup[colour][piece]
        return zobrist_table[index][piece_index]

    def get_rotation_hash(self, index, rotation):
        """
        Gets the random value for 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`

```
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.
```

```
39
40          Raises:
41              Exception: Invalid depth or score.
42          """
43          if depth == 0 or alpha < score < beta:
44              flag = TranspositionFlag.EXACT
45              score = score
46          elif score <= alpha:
47              flag = TranspositionFlag.UPPER
48              score = alpha
49          elif score >= beta:
50              flag = TranspositionFlag.LOWER
51              score = beta
52          else:
53              raise Exception('(TranspositionTable.insert_entry)')
54
55          self._table[self.calculate_entry_index(hash_key)] = TranspositionEntry(
    score, move, flag, hash_key, depth)
56
57          if len(self._table) > self._max_entries:
58              # Removes the longest-existing entry to free up space for more up-to-
    date entries
59              # Expression to remove leftmost item taken from https://docs.python.
    org/3/library/collections.html#ordereddict-objects
60              (k := next(iter(self._table)), self._table.pop(k))
61
62      def get_entry(self, hash_key, depth, alpha, beta):
63          """
64          Gets an entry from the transposition table.
65
66          Args:
67              hash_key (int): The Zobrist hash key.
68              depth (int): The depth of the search.
69              alpha (int): The alpha value for pruning.
70              beta (int): The beta value for pruning.
71
72          Returns:
73              tuple[int, Move] | tuple[None, None]: The evaluation score and the
    best move found, if entry exists.
74          """
75          index = self.calculate_entry_index(hash_key)
76
77          if index not in self._table:
78              return None, None
79
80          entry = self._table[index]
81
82          if entry.hash_key == hash_key and entry.depth >= depth:
83              if entry.flag == TranspositionFlag.EXACT:
84                  return entry.score, entry.move
85
86              if entry.flag == TranspositionFlag.LOWER and entry.score >= beta:
87                  return entry.score, entry.move
88
89              if entry.flag == TranspositionFlag.UPPER and entry.score <= alpha:
90                  return entry.score, entry.move
91
92          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
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()

```

```
40          window.clear_apply_arguments(ShaderType.BLOOM)
41          window.clear_effect(ShaderType.RAYS)
42
43          return None
44
45      def startup(self, persist):
46          """
47          Startup function. Initialises all objects, widgets and game data.
48
49          Args:
50              persist (dict): Dict containing game entry data.
51          """
52          super().startup(REVIEW_WIDGETS, MUSIC['review'])
53
54          window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.
    BACKGROUND_WAVES)
55          window.set_apply_arguments(ShaderType.BLOOM, highlight_colours=[(pygame.
    Color('0x95e0cc')).rgb, pygame.Color('0xf14e52').rgb], colour_intensity=0.8)
56          REVIEW_WIDGETS['help'].kill()
57
58          self._moves = deque(GameEntry.parse_moves(persist.pop('moves', '')))
59          self._popped_moves = deque()
60          self._game_info = persist
61
62          self._board = Board(self._game_info['start_fen_string'])
63          self._piece_group = PieceGroup()
64          self._laser_draw = LaserDraw(self.board_position, self.board_size)
65          self._capture_draw = CaptureDraw(self.board_position, self.board_size)
66
67          self.initialise_widgets()
68          self.simulate_all_moves()
69          self.refresh_pieces()
70          self.refresh_widgets()
71
72          self.draw()
73
74      @property
75      def board_position(self):
76          return REVIEW_WIDGETS['chessboard'].position
77
78      @property
79      def board_size(self):
80          return REVIEW_WIDGETS['chessboard'].size
81
82      @property
83      def square_size(self):
84          return self.board_size[0] / 10
85
86      def initialise_widgets(self):
87          """
88          Initializes the widgets for a new game.
89          """
90          REVIEW_WIDGETS['move_list'].reset_move_list()
91          REVIEW_WIDGETS['move_list'].kill()
92          REVIEW_WIDGETS['scroll_area'].set_image()
93
94          REVIEW_WIDGETS['winner_text'].set_text(f'WINNER: {get_winner_string(self.
    _game_info["winner"])}')
95          REVIEW_WIDGETS['blue_piece_display'].reset_piece_list()
96          REVIEW_WIDGETS['red_piece_display'].reset_piece_list()
97
98          if self._game_info['time_enabled']:
```

```
99              REVIEW_WIDGETS['timer_disabled_text'].kill()
100         else:
101             REVIEW_WIDGETS['blue_timer'].kill()
102             REVIEW_WIDGETS['red_timer'].kill()
103 
104     def refresh_widgets(self):
105         """
106         Refreshes the widgets after every move.
107         """
108         REVIEW_WIDGETS['move_number_text'].set_text(f'MOVE NO: {(len(self._moves))
     / 2:.1f} / {(len(self._moves) + len(self._popped_moves)) / 2:.1f}')
109         REVIEW_WIDGETS['move_colour_text'].set_text(f'{self.calculate_colour().
    name} TO MOVE')
110 
111         if self._game_info['time_enabled']:
112             if len(self._moves) == 0:
113                 REVIEW_WIDGETS['blue_timer'].set_time(float(self._game_info['time'
    ]) * 60 * 1000)
114                 REVIEW_WIDGETS['red_timer'].set_time(float(self._game_info['time'
    ]) * 60 * 1000)
115             else:
116                 REVIEW_WIDGETS['blue_timer'].set_time(float(self._moves[-1]['
    blue_time']) * 60 * 1000)
117                 REVIEW_WIDGETS['red_timer'].set_time(float(self._moves[-1]['
    red_time']) * 60 * 1000)
118 
119         REVIEW_WIDGETS['scroll_area'].set_image()
120 
121     def refresh_pieces(self):
122         """
123         Refreshes the pieces on the board.
124         """
125         self._piece_group.initialise_pieces(self._board.get_piece_list(), self.
    board_position, self.board_size)
126 
127     def simulate_all_moves(self):
128         """
129         Simulates all moves at the start of every game to obtain laser results and
     fill up piece display and move list widgets.
130         """
131         for index, move_dict in enumerate(self._moves):
132             laser_result = self._board.apply_move(move_dict['move'], fire_laser=
    True)
133             self._moves[index]['laser_result'] = laser_result
134 
135             if laser_result.hit_square_bitboard:
136                 if laser_result.piece_colour == Colour.BLUE:
137                     REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.
    piece_hit)
138                 elif laser_result.piece_colour == Colour.RED:
139                     REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.
    piece_hit)
140 
141             REVIEW_WIDGETS['move_list'].append_to_move_list(move_dict['
    unparsed_move'])
142 
143     def calculate_colour(self):
144         """
145         Calculates the current active colour to move.
146 
147         Returns:
148             Colour: The current colour to move.
```

```
149         """
150         if self._game_info['start_fen_string'][-1].lower() == 'b':
151             initial_colour = Colour.BLUE
152         elif self._game_info['start_fen_string'][-1].lower() == 'r':
153             initial_colour = Colour.RED
154 
155         if len(self._moves) % 2 == 0:
156             return initial_colour
157         else:
158             return initial_colour.get_flipped_colour()
159 
160     def handle_move(self, move, add_piece=True):
161         """
162         Handles applying or undoing a move.
163 
164         Args:
165             move (dict): The move to handle.
166             add_piece (bool): Whether to add the captured piece to the display.
    Defaults to True.
167         """
168         laser_result = move['laser_result']
169         active_colour = self.calculate_colour()
170         self._laser_draw.add_laser(laser_result, laser_colour=active_colour)
171 
172         if laser_result.hit_square_bitboard:
173             if laser_result.piece_colour == Colour.BLUE:
174                 if add_piece:
175                     REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.
    piece_hit)
176                 else:
177                     REVIEW_WIDGETS['red_piece_display'].remove_piece(laser_result.
    piece_hit)
178             elif laser_result.piece_colour == Colour.RED:
179                 if add_piece:
180                     REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.
    piece_hit)
181                 else:
182                     REVIEW_WIDGETS['blue_piece_display'].remove_piece(laser_result
    .piece_hit)
183 
184             self._capture_draw.add_capture(
185                 laser_result.piece_hit,
186                 laser_result.piece_colour,
187                 laser_result.piece_rotation,
188                 bitboard_to_coords(laser_result.hit_square_bitboard),
189                 laser_result.laser_path[0][0],
190                 active_colour,
191                 shake=False
192             )
193 
194     def update_laser_mask(self):
195         """
196         Updates the laser mask for the light rays effect.
197         """
198         temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
199         self._piece_group.draw(temp_surface)
200         mask = pygame.mask.from_surface(temp_surface, threshold=127)
201         mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255,
    0, 0, 255))
202 
203         window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
204 
```

```
205     def get_event(self, event):
206         """
207         Processes Pygame events.
208 
209         Args:
210             event (pygame.event.Event): The event to handle.
211         """
212         if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
213             REVIEW_WIDGETS['help'].kill()
214 
215         widget_event = self._widget_group.process_event(event)
216 
217         if widget_event is None:
218             return
219 
220         match widget_event.type:
221             case None:
222                 return
223 
224             case ReviewEventType.MENU_CLICK:
225                 self.next = 'menu'
226                 self.done = True
227 
228             case ReviewEventType.PREVIOUS_CLICK:
229                 if len(self._moves) == 0:
230                     return
231 
232                 # Pop last applied move off first stack
233                 move = self._moves.pop()
234                 # Pushed onto second stack
235                 self._popped_moves.append(move)
236 
237                 # Undo last applied move
238                 self._board.undo_move(move['move'], laser_result=move['
    laser_result'])
239                 self.handle_move(move, add_piece=False)
240                 REVIEW_WIDGETS['move_list'].pop_from_move_list()
241 
242                 self.refresh_pieces()
243                 self.refresh_widgets()
244                 self.update_laser_mask()
245 
246             case ReviewEventType.NEXT_CLICK:
247                 if len(self._popped_moves) == 0:
248                     return
249 
250                 # Peek at second stack to get last undone move
251                 move = self._popped_moves[-1]
252 
253                 # Reapply last undone move
254                 self._board.apply_move(move['move'])
255                 self.handle_move(move, add_piece=True)
256                 REVIEW_WIDGETS['move_list'].append_to_move_list(move['
    unparsed_move'])
257 
258                 # Pop last undone move from second stack
259                 self._popped_moves.pop()
260                 # Push onto first stack
261                 self._moves.append(move)
262 
263                 self.refresh_pieces()
264                 self.refresh_widgets()
```

```
265                 self.update_laser_mask()
266
267             case ReviewEventType.HELP_CLICK:
268                 self._widget_group.add(REVIEW_WIDGETS['help'])
269                 self._widget_group.handle_resize(window.size)
270
271     def handle_resize(self):
272         """
273         Handles resizing of the window.
274         """
275         super().handle_resize()
276         self._piece_group.handle_resize(self.board_position, self.board_size)
277         self._laser_draw.handle_resize(self.board_position, self.board_size)
278         self._capture_draw.handle_resize(self.board_position, self.board_size)
279
280         if self._laser_draw.firing:
281             self.update_laser_mask()
282
283     def draw(self):
284         """
285         Draws all components onto the window screen.
286         """
287         self._capture_draw.update()
288         self._widget_group.draw()
289         self._piece_group.draw(window.screen)
290         self._laser_draw.draw(window.screen)
291         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`

```
1  import sqlite3
2  from pathlib import Path
3
4  database_path = (Path(__file__).parent / '../database.db').resolve()
5
6  def upgrade():
7      """
8      Upgrade function to create games table.
9      """
10     connection = sqlite3.connect(database_path)
11     cursor = connection.cursor()
12
13     cursor.execute('''
14         CREATE TABLE games(
15             id INTEGER PRIMARY KEY,
16             cpu_enabled INTEGER NOT NULL,
17             cpu_depth INTEGER,
18             winner INTEGER,
19             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

```
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()
```

```
33
34 upgrade()
35 # 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

```
1  import sqlite3
2  from pathlib import Path
3  from datetime import datetime
4
5  database_path = (Path(__file__).parent / '../database/database.db').resolve()
6
7  def insert_into_games(game_entry):
8      """
9      Inserts a new row into games table.
10
11     Args:
12         game_entry (GameEntry): GameEntry object containing game information.
13     """
14     connection = sqlite3.connect(database_path, detect_types=sqlite3.
       PARSE_DECLTYPES)
15     connection.row_factory = sqlite3.Row
16     cursor = connection.cursor()
17
18     # Datetime added for created_dt column
19     game_entry = (*game_entry, datetime.now())
20
21     cursor.execute('''
22         INSERT INTO games (cpu_enabled, cpu_depth, winner, time_enabled, time,
       number_of_ply, moves, start_fen_string, final_fen_string, created_dt)
23         VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
24     ''', game_entry)
25
26     connection.commit()
27
28     # Return inserted row
29     cursor.execute('''
30         SELECT * FROM games WHERE id = LAST_INSERT_ROWID()
31     ''')
32     inserted_row = cursor.fetchone()
33
34     connection.close()
35
36     return dict(inserted_row)
37
38 def get_all_games():
39     """
40     Get all rows in games table.
41
42     Returns:
43         list[dict]: List of game entries represented as dictionaries.
44     """
45     connection = sqlite3.connect(database_path, detect_types=sqlite3.
       PARSE_DECLTYPES)
46     connection.row_factory = sqlite3.Row
47     cursor = connection.cursor()
```

```
48
49      cursor.execute('''
50          SELECT * FROM games
51      ''')
52      games = cursor.fetchall()
53
54      connection.close()
55
56      return [dict(game) for game in games]
57
58  def delete_all_games():
59      """
60      Delete all rows in games table.
61      """
62      connection = sqlite3.connect(database_path)
63      cursor = connection.cursor()
64
65      cursor.execute('''
66          DELETE FROM games
67      ''')
68
69      connection.commit()
70      connection.close()
71
72  def delete_game(id):
73      """
74      Deletes specific row in games table using id attribute.
75
76      Args:
77          id (int): Primary key for row.
78      """
79      connection = sqlite3.connect(database_path)
80      cursor = connection.cursor()
81
82      cursor.execute('''
83          DELETE FROM games WHERE id = ?
84      ''', (id,))
85
86      connection.commit()
87      connection.close()
88
89  def get_ordered_games(column, ascend=True, start_row=1, end_row=10):
90      """
91      Get specific number of rows from games table ordered by a specific column(s).
92
93      Args:
94          column (_type_): Column to sort by.
95          ascend (bool, optional): Sort ascending or descending. Defaults to True.
96          start_row (int, optional): First row returned. Defaults to 1.
97          end_row (int, optional): Last row returned. Defaults to 10.
98
99      Raises:
100         ValueError: If ascend argument or column argument are invalid types.
101
102     Returns:
103         list[dict]: List of ordered game entries represented as dictionaries.
104     """
105     if not isinstance(ascend, bool) or not isinstance(column, str):
106         raise ValueError('(database_helpers.get_ordered_games) Invalid input
        arguments!')
107
```

```
    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.

Each fragment shader is written in GLSL and stored in a `.frag` file, and read into the `ShaderManager` class.

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 = {}
```

```
45          self._frag_shaders = {}
46          self._programs = {}
47          self._vaos = {}
48          self._textures = {}
49          self._shader_passes = {}
50          self.framebuffers = {}
51
52          self.load_shader(ShaderType.BASE)
53          self.load_shader(ShaderType._CALIBRATE)
54          self.create_framebuffer(ShaderType._CALIBRATE)
55
56      def load_shader(self, shader_type, **kwargs):
57          """
58          Loads a given shader by creating a VAO reading the corresponding .frag
    file.
59
60          Args:
61              shader_type (ShaderType): The type of shader to load.
62              **kwargs: Additional arguments passed when initialising the fragment
    shader class.
63          """
64          self._shader_passes[shader_type] = shader_pass_lookup[shader_type](self,
    **kwargs)
65          self.create_vao(shader_type)
66
67      def clear_shaders(self):
68          """
69          Clears the shader list, leaving only the base shader.
70          """
71          self._shader_list = [ShaderType.BASE]
72
73      def create_vao(self, shader_type):
74          """
75          Creates a vertex array object (VAO) for the given shader type.
76
77          Args:
78              shader_type (ShaderType): The type of shader.
79          """
80          frag_name = shader_type[1:] if shader_type[0] == '_' else shader_type
81          vert_path = Path(shader_path / 'vertex/base.vert').resolve()
82          frag_path = Path(shader_path / f'fragments/{frag_name}.frag').resolve()
83
84          self._vert_shaders[shader_type] = vert_path.read_text()
85          self._frag_shaders[shader_type] = frag_path.read_text()
86
87          program = self._ctx.program(vertex_shader=self._vert_shaders[shader_type],
     fragment_shader=self._frag_shaders[shader_type])
88          self._programs[shader_type] = program
89
90          if shader_type == ShaderType._CALIBRATE:
91              self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._pygame_buffer, '2f 2f', 'vert', 'texCoords')])
92          else:
93              self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._opengl_buffer, '2f 2f', 'vert', 'texCoords')])
94
95      def create_framebuffer(self, shader_type, size=None, filter=moderngl.NEAREST):
96          """
97          Creates a framebuffer for the given shader type.
98
99          Args:
100             shader_type (ShaderType): The type of shader.
```

```
101             size (tuple[int, int], optional): The size of the framebuffer.
    Defaults to screen size.
102             filter (moderngl.Filter, optional): The texture filter. Defaults to
    NEAREST.
103         """
104         texture_size = size or self._screen_size
105         texture = self._ctx.texture(size=texture_size, components=4)
106         texture.filter = (filter, filter)
107 
108         self._textures[shader_type] = texture
109         self.framebuffers[shader_type] = self._ctx.framebuffer(color_attachments=[
    self._textures[shader_type]])
110 
111     def render_to_fbo(self, shader_type, texture, output_fbo=None, program_type=
    None, use_image=True, **kwargs):
112         """
113         Applies the shaders and renders the resultant texture to a framebuffer
    object (FBO).
114 
115         Args:
116             shader_type (ShaderType): The type of shader.
117             texture (moderngl.Texture): The texture to render.
118             output_fbo (moderngl.Framebuffer, optional): The output framebuffer.
    Defaults to None.
119             program_type (ShaderType, optional): The program type. Defaults to
    None.
120             use_image (bool, optional): Whether to use the image uniform. Defaults
     to True.
121             **kwargs: Additional uniforms for the fragment shader.
122         """
123         fbo = output_fbo or self.framebuffers[shader_type]
124         program = self._programs[program_type] if program_type else self._programs
    [shader_type]
125         vao= self._vaos[program_type] if program_type else self._vaos[shader_type]
126 
127         fbo.use()
128         texture.use(0)
129 
130         if use_image:
131             program['image'] = 0
132         for uniform, value in kwargs.items():
133             program[uniform] = value
134 
135         vao.render(mode=moderngl.TRIANGLE_STRIP)
136 
137     def apply_shader(self, shader_type, **kwargs):
138         """
139         Applies a shader of the given type and adds it to the list.
140 
141         Args:
142             shader_type (ShaderType): The type of shader to apply.
143 
144         Raises:
145             ValueError: If the shader is already being applied.
146         """
147         if shader_type in self._shader_list:
148             return
149 
150         self.load_shader(shader_type, **kwargs)
151         self._shader_list.append(shader_type)
152 
153         # Sort shader list based on the order in SHADER_PRIORITY, so that more
```

```
        important shaders are applied first
154         self._shader_list.sort(key=lambda shader: -SHADER_PRIORITY.index(shader))
155
156     def remove_shader(self, shader_type):
157         """
158         Removes a shader of the given type from the list.
159
160         Args:
161             shader_type (ShaderType): The type of shader to remove.
162         """
163         if shader_type in self._shader_list:
164             self._shader_list.remove(shader_type)
165
166     def render_output(self):
167         """
168         Renders the final output to the screen.
169         """
170         # Render to the screen framebuffer
171         self._ctx.screen.use()
172
173         # Take the texture of the last framebuffer to be rendered to, and render
        that to the screen framebuffer
174         output_shader_type = self._shader_list[-1]
175         self.get_fbo_texture(output_shader_type).use(0)
176         self._programs[output_shader_type]['image'] = 0
177
178         self._vaos[output_shader_type].render(mode=moderngl.TRIANGLE_STRIP)
179
180     def get_fbo_texture(self, shader_type):
181         """
182         Gets the texture from the specified shader type's FBO.
183
184         Args:
185             shader_type (ShaderType): The type of shader.
186
187         Returns:
188             moderngl.Texture: The texture from the FBO.
189         """
190         return self.framebuffers[shader_type].color_attachments[0]
191
192     def calibrate_pygame_surface(self, pygame_surface):
193         """
194         Converts the Pygame window surface into an OpenGL texture.
195
196         Args:
197             pygame_surface (pygame.Surface): The finished Pygame surface.
198
199         Returns:
200             moderngl.Texture: The calibrated texture.
201         """
202         texture = self._ctx.texture(pygame_surface.size, 4)
203         texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
204         texture.swizzle = 'BGRA'
205         # Take the Pygame surface's pixel array and draw it to the new texture
206         texture.write(pygame_surface.get_view('1'))
207
208         # ShaderType._CALIBRATE has a VAO containing the pygame_quad_array
        coordinates, as Pygame uses different texture coordinates than ModernGL
        textures
209         self.render_to_fbo(ShaderType._CALIBRATE, texture)
210         return self.get_fbo_texture(ShaderType._CALIBRATE)
211
```

```
212     def draw(self, surface, arguments):
213         """
214         Draws the Pygame surface with shaders applied to the screen.
215 
216         Args:
217             surface (pygame.Surface): The final Pygame surface.
218             arguments (dict): A dict of { ShaderType: Args } items, containing
    keyword arguments for every fragment shader.
219         """
220         self._ctx.viewport = (0, 0, *self._screen_size)
221         texture = self.calibrate_pygame_surface(surface)
222 
223         for shader_type in self._shader_list:
224             self._shader_passes[shader_type].apply(texture, **arguments.get(
    shader_type, {}))
225             texture = self.get_fbo_texture(shader_type)
226 
227         self.render_output()
228 
229     def __del__(self):
230         """
231         Cleans up ModernGL resources when the ShaderManager object is deleted.
232         """
233         self.cleanup()
234 
235     def cleanup(self):
236         """
237         Cleans up resources used by the ModernGL.
238         Probably unnecessary as the 'auto' garbage collection mode is used.
239         """
240         self._pygame_buffer.release()
241         self._opengl_buffer.release()
242         for program in self._programs:
243             self._programs[program].release()
244         for texture in self._textures:
245             self._textures[texture].release()
246         for vao in self._vaos:
247             self._vaos[vao].release()
248         for framebuffer in self.framebuffers:
249             self.framebuffers[framebuffer].release()
250 
251     def handle_resize(self, new_screen_size):
252         """
253         Handles resizing of the screen.
254 
255         Args:
256             new_screen_size (tuple[int, int]): The new screen size.
257         """
258         self._screen_size = new_screen_size
259 
260         # Recreate all framebuffers to prevent scaling issues
261         for shader_type in self.framebuffers:
262             filter = self._textures[shader_type].filter[0]
263             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

```
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

```
// 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)

    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)
```

```
44
45          # Extract bright colours (highlights) from the texture
46          _HighlightBrightness(self._shader_manager).apply(texture, intensity=
    brightness_intensity)
47          highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
    _HIGHLIGHT_BRIGHTNESS)
48
49          # Use colour as highlights
50          for colour in highlight_colours:
51              _HighlightColour(self._shader_manager).apply(texture, old_highlight=
    highlight_texture, colour=colour, intensity=colour_intensity)
52              highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
    _HIGHLIGHT_COLOUR)
53
54          # Apply Gaussian blur to highlights
55          _Blur(self._shader_manager).apply(highlight_texture)
56
57          # Add the pixel values for the highlights onto the base texture
58          self._shader_manager.get_fbo_texture(ShaderType._BLUR).use(1)
59          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`

```
1  # version 330 core
2
3  in vec2 uvs;
4  out vec4 f_colour;
5
6  uniform sampler2D image;
7  uniform vec3 checkColour;
8
9  void main() {
10     vec4 pixel = texture(image, uvs);
11
12     // If pixel is occluding colour, set pixel to white
13     if (pixel.rgb == checkColour) {
14         f_colour = vec4(1.0, 1.0, 1.0, 1.0);
15     // Else, set pixel to black
16     } else {
17         f_colour = vec4(vec3(0.0), 1.0);
18     }
19 }
```

Shadowmap

The shadowmap fragment shader takes the occluding texture and creates a 1D shadow map. The algorithm begins with assuming that all light rays are not occluded and able to reach the edge

of the texture, hence `maxDistance`, the furthest distance a light ray can travel from the centre for its angle, is set to 1.

As we sample further from the centre, before the light ray hits an occluding object, `maxDistance` does not change. When it first hits an occluding object, the `step` value on line 32 will be valid, and `maxDistance` will be set to the distance of the sampled pixel. Past this, `maxDistance` will remain the same, since both arguments in the `max` function will resolve to the current `maxDistance`. Hence, `maxDistance` will always be the distance from the centre to the nearest occluding pixel.

`shadowmap.frag`

```
1  # version 330 core
2
3  #define PI 3.1415926536;
4
5  in vec2 uvs;
6  out vec4 f_colour;
7
8  uniform sampler2D image;
9  uniform float resolution;
10 uniform float THRESHOLD=0.99;
11
12 void main() {
13   float maxDistance = 1.0;
14
15     for (float y = 0.0 ; y < resolution ; y += 1.0) {
16         //rectangular to polar filter
17         float currDistance = y / resolution;
18
19         vec2 norm = vec2(uvs.x, currDistance) * 2.0 - 1.0; // Range from [0, 1] ->
       [-1, 1]
20         float angle = (1.5 - norm.x) * PI; // Range from [-1, 1] -> [0.5PI, 2.5PI]
21         float radius = (1.0 + norm.y) * 0.5; // Range from [-1, 1] -> [0, 1]
22
23         //coord which we will sample from occlude map
24         vec2 coords = vec2(radius * -sin(angle), radius * -cos(angle)) / 2.0 +
     0.5;
25
26         // Sample occlusion map
27         vec4 occluding = texture(image, coords);
28
29         // If pixel is not occluding (Red channel value below threshold), set
     maxDistance to current distance
30     // If pixel is occluding, don't change distance
31     // maxDistance therefore is the distance from the center to the nearest
     occluding pixel
32         maxDistance = max(maxDistance * step(occluding.r, THRESHOLD), min(
     maxDistance, currDistance));
33     }
34
35     f_colour = vec4(vec3(maxDistance), 1.0);
36 }
```

Lightmap

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

`lightmap.frag`

```
1  # version 330 core
2
```

```
#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

```
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:
```

```
40                  # Calibrate Pygame mask surface and crop it
41                  occlusion_texture = self._shader_manager.calibrate_pygame_surface(
    occlusion)
42                  _Crop(self._shader_manager).apply(occlusion_texture, relative_pos=
    light_topleft, relative_size=relative_size)
43                  occlusion_texture = self._shader_manager.get_fbo_texture(
    ShaderType._CROP)
44              else:
45                  occlusion_texture = None
46
47              # Apply lightmap shader, shadowmap and occlusion are included within
    the _Lightmap class
48              _Lightmap(self._shader_manager).apply(cropped_texture, colour,
    softShadow, occlusion_texture, *args)
49              light_map = self._shader_manager.get_fbo_texture(ShaderType._LIGHTMAP)
50
51              # Blend the final result with the original texture
52              _Blend(self._shader_manager).apply(final_texture, light_map,
    light_topleft)
53              final_texture = self._shader_manager.get_fbo_texture(ShaderType._BLEND
    )
54
55          self._shader_manager.render_to_fbo(ShaderType.RAYS, final_texture)
```