Chapter 1

Technical Solution

1.1 File Tree Diagram

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

```
resources/ ································· Asset files used in program
data/
      database/
                  migrations/ ··············· Database migrations that track changes to table schema
                  database.db
      shaders/
                  classes/ ················· Python classes for each fragment shader
                  fragments/ ··············· Fragment shader code
                  vertex/ ················· Vertex shader code
      widgets/
                  bases/ ················· Abstract classes for widgets
                  ...
      states/
                  game/
                              components/ ··· Collection of classes involved in running of game
                                              screen
                              cpu/
                              mvc/
                              game.py
                              widget_dict.py· Initalises a dictionary containing widgets for the game screen
                                              GUI
                  ...
      app_data/ ························· JSON files used to store settings and configurations
      components/ ························· Collection of useful classes used in multiple areas
      managers/ ························· Collection of classes that manage game systems
      utils/ ····························· Collection of helper functions for classes
      main.py
      setup.py
      assets.py
      control.py
      constants.py
      loading_screen.py
      windows_setup.py
```
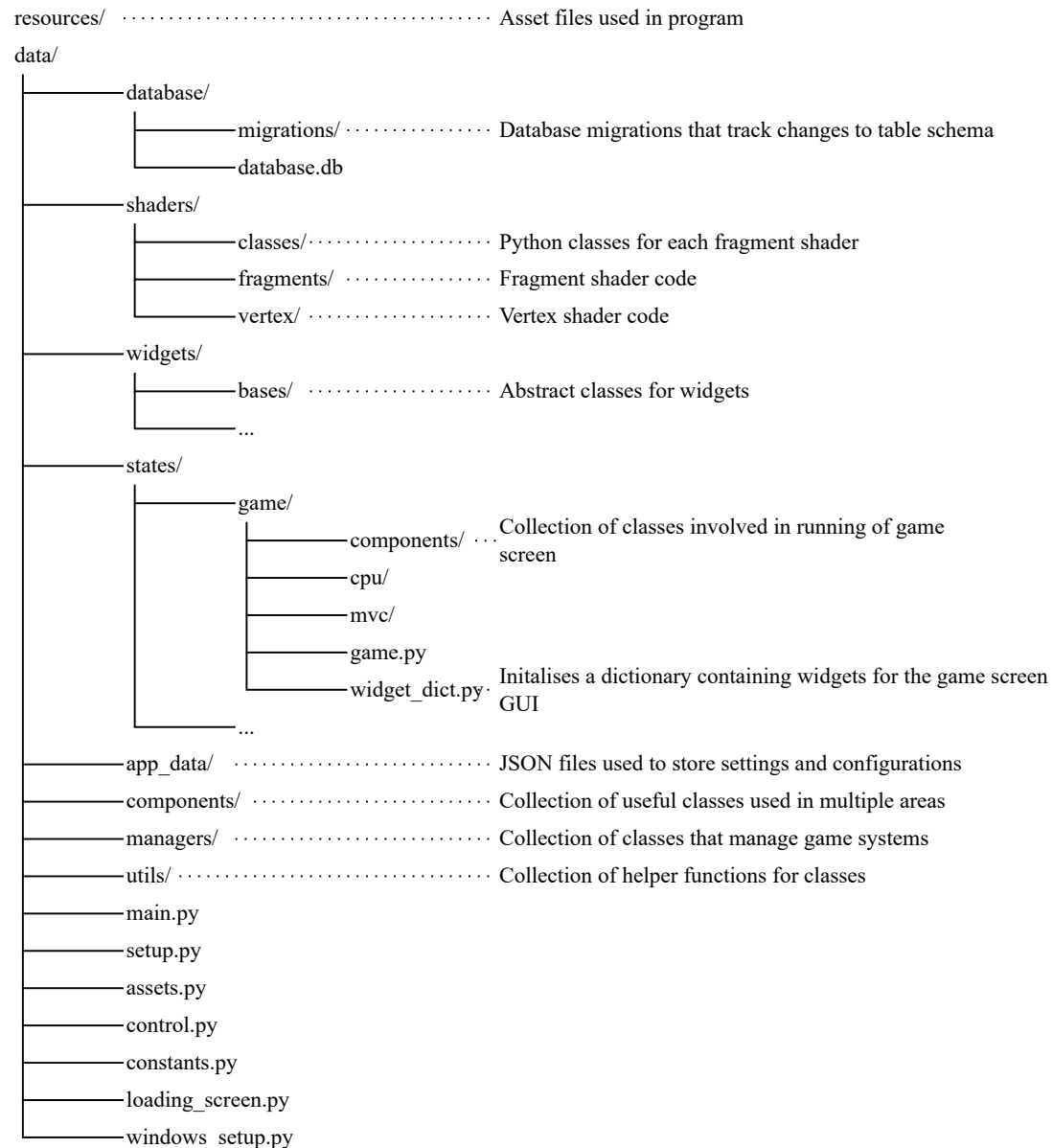
Figure 1.1: File tree diagram

1.2 Summary of Complexity

- Alpha-beta pruning and transposition table improvements for Minimax (1.6.2 and 1.6.3)

- 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 1.6.5)

- Bitboards (1.5.5)

- Zobrist hashing (1.6.6)

- (File handling and JSON parsing) (1.3.3)

- (Dictionary recursion) (1.3.4)

- (Dot product) (1.3.3 and 1.9.2)

1.3 Overview

1.3.1 Main

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

`main.py`

```python
1  from sys import platform
2  # Initialises Pygame
3  import data.setup
4
5  # Windows OS requires some configuration for Pygame to scale GUI continuously
       while window is being resized
6  if platform == 'win32':
7      import data.windows_setup as win_setup
8
9  from data.loading_screen import LoadingScreen
10
11 states = [None, None]
12
13 def load_states():
14     """
15     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
16     """
17     from data.control import Control
18     from data.states.game.game import Game
19     from data.states.menu.menu import Menu
```

```
20      from data.states.settings.settings import Settings
21      from data.states.config.config import Config
22      from data.states.browser.browser import Browser
23      from data.states.review.review import Review
24      from data.states.editor.editor import Editor
25
26      state_dict = {
27          'menu': Menu(),
28          'game': Game(),
29          'settings': Settings(),
30          'config': Config(),
31          'browser': Browser(),
32          'review': Review(),
33          'editor': Editor()
34      }
35
36      app = Control()
37
38      states[0] = app
39      states[1] = state_dict
40
41  loading_screen = LoadingScreen(load_states)
42
43  def main():
44      """
45      Executed by run.py, starts main game loop
46      """
47      app, state_dict = states
48
49      if platform == 'win32':
50          win_setup.set_win_resize_func(app.update_window)
51
52      app.setup_states(state_dict, 'menu')
53      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

```
1  import pygame
2  import threading
3  import sys
4  from pathlib import Path
5  from data.utils.load_helpers import load_gfx, load_sfx
6  from data.managers.window import window
7  from data.managers.audio import audio
8
9  FPS = 30
10 start_ticks = pygame.time.get_ticks()
11 logo_gfx_path = (Path(__file__).parent / '../resources/graphics/gui/icons/logo/
       logo.png').resolve()
12 sfx_path_1 = (Path(__file__).parent / '../resources/sfx/loading_screen/
       loading_screen_1.wav').resolve()
13 sfx_path_2 = (Path(__file__).parent / '../resources/sfx/loading_screen/
       loading_screen_2.wav').resolve()
14
15 def easeOutBack(progress):
```

```
16        """
17        Represents a cubic function for easing the logo position.
18        Starts quickly and has small overshoot, then ends slowly.
19
20        Args:
21            progress (float): x-value for cubic function ranging from 0-1.
22
23        Returns:
24            float: 2.70x^3 + 1.70x^2 + 0x + 1, where x is time elapsed.
25        """
26        c2 = 1.70158
27        c3 = 2.70158
28
29        return c3 * ((progress - 1) ** 3) + c2 * ((progress - 1) ** 2) + 1
30
31 class LoadingScreen:
32     def __init__(self, target_func):
33         """
34         Creates new thread, and sets the load_state() function as its target.
35         Then starts draw loop for the loading screen.
36
37         Args:
38             target_func (Callable): function to be run on thread.
39         """
40         self._clock = pygame.time.Clock()
41         self._thread = threading.Thread(target=target_func)
42         self._thread.start()
43
44         self._logo_surface = load_gfx(logo_gfx_path)
45         self._logo_surface = pygame.transform.scale(self._logo_surface, (96, 96))
46         audio.play_sfx(load_sfx(sfx_path_1))
47         audio.play_sfx(load_sfx(sfx_path_2))
48
49         self.run()
50
51     @property
52     def logo_position(self):
53         duration = 1000
54         displacement = 50
55         elapsed_ticks = pygame.time.get_ticks() - start_ticks
56         progress = min(1, elapsed_ticks / duration)
57         center_pos = ((window.screen.size[0] - self._logo_surface.size[0]) / 2, (
58 window.screen.size[1] - self._logo_surface.size[1]) / 2)
59
60         return (center_pos[0], center_pos[1] + displacement - displacement *
61 easeOutBack(progress))
62
63     @property
64     def logo_opacity(self):
65         return min(255, (pygame.time.get_ticks() - start_ticks) / 5)
66
67     @property
68     def duration_not_over(self):
69         return (pygame.time.get_ticks() - start_ticks) < 1500
70
71     def event_loop(self):
72         """
73         Handles events for the loading screen, no user input is taken except to
74 quit the game.
75         """
76         for event in pygame.event.get():
77             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 sample, 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     """
```

```python
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
62        around a rectangular image surface's perimeter.
63
64        Args:
65            image_size (tuple[float, float]): Image surface size.
66            number (int): Number of points to be generated.
67
68        Returns:
69            list[tuple[int, int], ...]: List of random points on perimeter of image
        surface.
70        """
71        perimeter = 2 * (image_size[0] + image_size[1])
72        # 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
73        perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in
        range(0, number)]
74        pos_list = []
75
76        for perimeter_offset in perimeter_offsets:
77            # For every point, add a random offset
78            max_displacement = int(perimeter / (number * 4))
79            perimeter_offset += randint(-max_displacement, max_displacement)
80
81            if perimeter_offset > perimeter:
82                perimeter_offset -= perimeter
83
84            # Convert 1D distance back into 2D points on image surface perimeter
85            if perimeter_offset < image_size[0]:
86                pos_list.append((perimeter_offset, 0))
87            elif perimeter_offset < image_size[0] + image_size[1]:
88                pos_list.append((image_size[0], perimeter_offset - image_size[0]))
89            elif perimeter_offset < image_size[0] + image_size[1] + image_size[0]:
90                pos_list.append((perimeter_offset - image_size[0] - image_size[1],
```

```python
              image_size [1]))
90              else:
91                  pos_list.append((0, perimeter - perimeter_offset))
92      return pos_list

93
94  def get_angle_between_vectors(u, v, deg=True):
95      """
96      Uses the dot product formula to find the angle between two vectors.
97
98      Args:
99          u (list[int, int]): Vector 1.
100         v (list[int, int]): Vector 2.
101         deg (bool, optional): Return results in degrees. Defaults to True.
102
103     Returns:
104         float: Angle between vectors.
105     """
106     dot_product = sum(i * j for (i, j) in zip(u, v))
107     u_magnitude = math.sqrt(u[0] ** 2 + u[1] ** 2)
108     v_magnitude = math.sqrt(v[0] ** 2 + v[1] ** 2)
109
110     cos_angle = dot_product / (u_magnitude * v_magnitude)
111     radians = math.acos(min(max(cos_angle, -1), 1))
112
113     if deg:
114         return math.degrees(radians)
115     else:
116         return radians
117
118  def get_rotational_angle(u, v, deg=True):
119     """
120     Get bearing angle relative to positive x-axis centered on second vector.
121
122     Args:
123         u (list[int, int]): Vector 1.
124         v (list[int, int]): Vector 2, set as center of axes.
125         deg (bool, optional): Return results in degrees. Defaults to True.
126
127     Returns:
128         float: Bearing angle between vectors.
129     """
130     radians = math.atan2(u[1] - v[1], u[0] -v[0])
131
132     if deg:
133         return math.degrees(radians)
134     else:
135         return radians
136
137  def get_vector(src_vertex, dest_vertex):
138     """
139     Get vector describing translation between two points.
140
141     Args:
142         src_vertex (list[int, int]): Source vertex.
143         dest_vertex (list[int, int]): Destination vertex.
144
145     Returns:
146         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):
```

8

```python
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
205         Args :
206             screen ( pygame . Surface ): Screen to be drawn to
207             background ( list [ pygame . Surface , ...] | pygame . Surface ): Background to be
            drawn , if GIF , list of surfaces indexed to select frame to be drawn
```

9

```
208        current_time (int, optional): Used to calculate frame index for GIF.
      Defaults to 0.
209        """
210    if isinstance(background, list):
211        # Animated background passed in as list of surfaces, calculate_frame_index
      () used to get index of frame to be drawn
212        frame_index = calculate_frame_index(current_time, 0, len(background), fps
      =8)
213        scaled_background = scale_and_cache(background[frame_index], screen.size)
214        screen.blit(scaled_background, (0, 0))
215    else:
216        scaled_background = scale_and_cache(background, screen.size)
217        screen.blit(scaled_background, (0, 0))
218
219 def get_highlighted_icon(icon):
220    """
221    Used for pressable icons, draws overlay on icon to show as pressed.
222
223    Args:
224        icon (pygame.Surface): Icon surface.
225
226    Returns:
227        pygame.Surface: Icon with overlay drawn on top.
228    """
229    icon_copy = icon.copy()
230    overlay = pygame.Surface((icon.get_width(), icon.get_height()), pygame.
      SRCALPHA)
231    overlay.fill((0, 0, 0, 128))
232    icon_copy.blit(overlay, (0, 0))
233    return icon_copy
```

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

```
1 import json
2 from pathlib import Path
3
4 module_path = Path(__file__).parent
5 default_file_path = (module_path / '../app_data/default_settings.json').resolve()
6 user_file_path = (module_path / '../app_data/user_settings.json').resolve()
7 themes_file_path = (module_path / '../app_data/themes.json').resolve()
8
9 def load_json(path):
10    """
11    Args:
12        path (str): Path to JSON file.
13
14    Raises:
15        Exception: Invalid file.
16
17    Returns:
18        dict: Parsed JSON file.
19    """
20    try:
21        with open(path, 'r') as f:
22            file = json.load(f)
23
24        return file
25    except:
26        raise Exception('Invalid JSON file (data_helpers.py)')
27
28 def get_user_settings():
```

```
29      return load_json(user_file_path)
30
31  def get_default_settings():
32      return load_json(default_file_path)
33
34  def get_themes():
35      return load_json(themes_file_path)
36
37  def update_user_settings(data):
38      """
39      Rewrites JSON file for user settings with new data.
40
41      Args:
42          data (dict): Dictionary storing updated user settings.
43
44      Raises:
45          Exception: Invalid file.
46      """
47      try:
48          with open(user_file_path, 'w') as f:
49              json.dump(data, f, indent=4)
50      except:
51          raise Exception('Invalid JSON file (data_helpers.py)')
```

widget_helpers.py (Files used for creating widgets)

```
1   import pygame
2   from math import sqrt
3
4   def create_slider(size, fill_colour, border_width, border_colour):
5       """
6       Creates surface for sliders.
7
8       Args:
9           size (list[int, int]): Image size.
10          fill_colour (pygame.Color): Fill (inner) colour.
11          border_width (float): Border width.
12          border_colour (pygame.Color): Border colour.
13
14      Returns:
15          pygame.Surface: Slider image surface.
16      """
17      gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
18      border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
        height))
19
20      # Draws rectangle with a border radius half of image height, to draw an
        rectangle with semicurclar cap (obround)
21      pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int
        (size[1] / 2))
22      pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
        border_width), border_radius=int(size[1] / 2))
23
24      return gradient_surface
25
26  def create_slider_gradient(size, border_width, border_colour):
27      """
28      Draws surface for colour slider, with a full colour gradient as fill colour.
29
30      Args:
31          size (list[int, int]): Image size.
```

```
32          border_width (float): Border width.
33          border_colour (pygame.Color): Border colour.
34
35      Returns:
36          pygame.Surface: Slider image surface.
37      """
38      gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
39
40      first_round_end = gradient_surface.height / 2
41      second_round_end = gradient_surface.width - first_round_end
42      gradient_y_mid = gradient_surface.height / 2
43
44      # Iterate through length of slider
45      for i in range(gradient_surface.width):
46          draw_height = gradient_surface.height
47
48          if i < first_round_end or i > second_round_end:
49              # Draw semicircular caps if x-distance less than or greater than
    radius of cap (half of image height)
50              distance_from_cutoff = min(abs(first_round_end - i), abs(i -
    second_round_end))
51              draw_height = calculate_gradient_slice_height(distance_from_cutoff,
    gradient_surface.height / 2)
52
53          # Get colour from distance from left side of slider
54          color = pygame.Color(0)
55          color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)
56
57          draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
58          draw_rect.center = (i, gradient_y_mid)
59
60          pygame.draw.rect(gradient_surface, color, draw_rect)
61
62      border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
    height))
63      pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
    border_width), border_radius=int(size[1] / 2))
64
65      return gradient_surface
66
67  def calculate_gradient_slice_height(distance, radius):
68      """
69      Calculate height of vertical slice of semicircular slider cap.
70
71      Args:
72          distance (float): x-distance from center of circle.
73          radius (float): Radius of semicircle.
74
75      Returns:
76          float: Height of vertical slice.
77      """
78      return sqrt(radius ** 2 - distance ** 2) * 2 + 2
79
80  def create_slider_thumb(radius, colour, border_colour, border_width):
81      """
82      Creates surface with bordered circle.
83
84      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
```

13

```
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.
theme.py

```
1  from data.utils.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.utils.board_helpers import coords_to_screen_pos
3   from data.constants import EMPTY_BB, ShaderType, Colour
4   from data.managers.animation import animation
5   from data.managers.window import window
6   from data.managers.audio import audio
7   from data.assets import GRAPHICS, SFX
8   from data.constants import LaserType
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
67          if laser_result.hit_square_bitboard != EMPTY_BB:
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              audio.play_sfx(SFX['piece_destroy'])
73
74          laser_path = [(coords, rotation, type) for (coords, dir), rotation, type
     in zip(laser_path, laser_rotation, laser_types)]
75          self._laser_lists.append((laser_path, laser_colour))
76
77          window.clear_effect(ShaderType.RAYS)
78          window.set_effect(ShaderType.RAYS, lights=laser_lights)
```

```
79          animation.set_timer(1000, self.remove_laser)
80
81          audio.play_sfx(SFX['laser_1'])
82          audio.play_sfx(SFX['laser_2'])
83
84      def remove_laser(self):
85          """
86          Removes a laser from the board.
87          """
88          self._laser_lists.pop(0)
89
90          if len(self._laser_lists) == 0:
91              window.clear_effect(ShaderType.RAYS)
92
93      def draw_laser(self, screen, laser_list, glow=True):
94          """
95          Draws every laser on the screen.
96
97          Args:
98              screen (pygame.Surface): The screen to draw on.
99              laser_list (list): The list of laser segments to draw.
100             glow (bool, optional): Whether to draw a glow effect. Defaults to True
    .
101         """
102         laser_path, laser_colour = laser_list
103         laser_list = []
104         glow_list = []
105
106         for coords, rotation, type in laser_path:
107             square_x, square_y = coords_to_screen_pos(coords, self._board_position
    , self._square_size)
108
109             image = GRAPHICS[type_to_image[type][laser_colour]]
110             rotated_image = pygame.transform.rotate(image, rotation.to_angle())
111             scaled_image = pygame.transform.scale(rotated_image, (self.
    _square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating
    black lines
112             laser_list.append((scaled_image, (square_x, square_y)))
113
114             # Scales up the laser image surface as a glow surface
115             scaled_glow = pygame.transform.scale(rotated_image, (self._square_size
     * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
116             offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
117             glow_list.append((scaled_glow, (square_x - offset, square_y - offset))
    )
118
119         # Scaled glow surfaces drawn on top with the RGB_ADD blend mode
120         if glow:
121             screen.fblits(glow_list, pygame.BLEND_RGB_ADD)
122
123         screen.blits(laser_list)
124
125     def draw(self, screen):
126         """
127         Draws all lasers on the screen.
128
129         Args:
130             screen (pygame.Surface): The screen to draw on.
131         """
132         for laser_list in self._laser_lists:
133             self.draw_laser(screen, laser_list)
134
```

17

```
135     def handle_resize(self, board_position, board_size):
136         """
137         Handles resizing of the board.
138
139         Args:
140             board_position (tuple[int, int]): The new position of the board.
141             board_size (tuple[int, int]): The new size of the board.
142         """
143         self._board_position = board_position
144         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.utils.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
6  class ParticlesDraw:
7      def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
8          self._particles = []
9          self._glow_particles = []
10
11         self._gravity = gravity
12         self._rotation = rotation
13         self._shrink = shrink
14         self._opacity = opacity
15
16     def fragment_image(self, image, number):
17         image_size = image.get_rect().size
18         """
19         1. Takes an image surface and samples random points on the perimeter.
20         2. Iterates through points, and depending on the nature of two consecutive
       points, finds a corner between them.
21         3. Draws a polygon with the points as the vertices to mask out the area
       not in the fragment.
22
23         Args:
24             image (pygame.Surface): Image to fragment.
25             number (int): The number of fragments to create.
26
27         Returns:
28             list[pygame.Surface]: List of image surfaces with fragment of original
       surface drawn on top.
29         """
30         center = image.get_rect().center
31         points_list = get_perimeter_sample(image_size, number)
32         fragment_list = []
33
34         points_list.append(points_list[0])
35
36         # Iterate through points_list, using the current point and the next one
37         for i in range(len(points_list) - 1):
38             vertex_1 = points_list[i]
```

18

```
39            vertex_2 = points_list[i + 1]
40            vector_1 = get_vector(center, vertex_1)
41            vector_2 = get_vector(center, vertex_2)
42            angle = get_angle_between_vectors(vector_1, vector_2)
43
44            cropped_image = pygame.Surface(image_size, pygame.SRCALPHA)
45            cropped_image.fill((0, 0, 0, 0))
46            cropped_image.blit(image, (0, 0))
47
48            corners_to_draw = None
49
50            if vertex_1[0] == vertex_2[0] or vertex_1[1] == vertex_2[1]: # Points
   on the same side
51                corners_to_draw = 4
52
53            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
54                corners_to_draw = 2
55
56            elif angle < 180: # Points on adjacent sides
57                corners_to_draw = 3
58
59            else:
60                corners_to_draw = 1
61
62            corners_list = []
63            for j in range(corners_to_draw):
64                if len(corners_list) == 0:
65                    corners_list.append(get_next_corner(vertex_2, image_size))
66                else:
67                    corners_list.append(get_next_corner(corners_list[-1],
   image_size))
68
69            pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *
   corners_list, vertex_1))
70
71            fragment_list.append(cropped_image)
72
73        return fragment_list
74
75    def add_captured_piece(self, piece, colour, rotation, position, size):
76        """
77        Adds a captured piece to fragment into particles.
78
79        Args:
80            piece (Piece): The piece type.
81            colour (Colour): The active colour of the piece.
82            rotation (int): The rotation of the piece.
83            position (tuple[int, int]): The position where particles originate
   from.
84            size (tuple[int, int]): The size of the piece.
85        """
86        piece_sprite = PieceSprite(piece, colour, rotation)
87        piece_sprite.set_geometry((0, 0), size)
88        piece_sprite.set_image()
89
90        particles = self.fragment_image(piece_sprite.image, 5)
91
92        for particle in particles:
93            self.add_particle(particle, position)
94
95    def add_sparks(self, radius, colour, position):
```

```python
        """
        Adds laser spark particles.

        Args:
            radius (int): The radius of the sparks.
            colour (Colour): The active colour of the sparks.
            position (tuple[int, int]): The position where particles originate
from.
        """
        for i in range(randint(10, 15)):
            velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
            random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in
colour]
            self._particles.append([None, [radius, random_colour], [*position],
velocity, 0])

    def add_particle(self, image, position):
        """
        Adds a particle.

        Args:
            image (pygame.Surface): The image of the particle.
            position (tuple): The position of the particle.
        """
        velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]

        # Each particle is stored with its attributes: [surface, copy of surface,
position, velocity, lifespan]
        self._particles.append([image, image.copy(), [*position], velocity, 0])

    def update(self):
        """
        Updates each particle and its attributes.
        """
        for i in range(len(self._particles) - 1, -1, -1):
            particle = self._particles[i]

            #update position
            particle[2][0] += particle[3][0]
            particle[2][1] += particle[3][1]

            #update lifespan
            self._particles[i][4] += 0.01

            if self._particles[i][4] >= 1:
                self._particles.pop(i)
                continue

            if isinstance(particle[1], pygame.Surface): # Particle is a piece
                # Update velocity
                particle[3][1] += self._gravity

                # Update size
                image_size = particle[1].get_rect().size
                end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink)
 * image_size[1])
                target_size = (image_size[0] - particle[4] * (image_size[0] -
end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))

                # Update rotation
                rotation = (self._rotation if particle[3][0] <= 0 else -self.
_rotation) * particle[4]
```

20

```
151
152                updated_image = pygame.transform.scale(pygame.transform.rotate(
        particle[1], rotation), target_size)
153
154            elif isinstance(particle[1], list): # Particle is a spark
155                # Update size
156                end_radius = (1 - self._shrink) * particle[1][0]
157                target_radius = particle[1][0] - particle[4] * (particle[1][0] -
        end_radius)
158
159                updated_image = pygame.Surface((target_radius * 2, target_radius *
         2), pygame.SRCALPHA)
160                pygame.draw.circle(updated_image, particle[1][1], (target_radius,
        target_radius), target_radius)
161
162            # Update opacity
163            alpha = 255 - particle[4] * (255 - self._opacity)
164
165            updated_image.fill((255, 255, 255, alpha), None, pygame.
        BLEND_RGBA_MULT)
166
167            particle[0] = updated_image
168
169    def draw(self, screen):
170        """
171        Draws the particles, indexing the surface and position attributes for each
         particle.
172
173        Args:
174            screen (pygame.Surface): The screen to draw on.
175        """
176        screen.blits([
177            (particle[0], particle[2]) for particle in self._particles
178        ])
```

1.4.3   Widget Bases

Widget bases are the base classes for for my widgets system. They contain both attributes and getter methods that provide basic functionality such as size and position, and abstract methods to be overriden. These bases are also 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`

```
1 import pygame
2 from data.constants import SCREEN_SIZE
3 from data.managers.theme import theme
4 from data.assets import DEFAULT_FONT
5
6 DEFAULT_SURFACE_SIZE = SCREEN_SIZE
7 REQUIRED_KWARGS = ['relative_position', 'relative_size']
8
```

```python
class _Widget(pygame.sprite.Sprite):
    def __init__(self, **kwargs):
        """
        Every widget has the following attributes:

        surface (pygame.Surface): The surface the widget is drawn on.
        raw_surface_size (tuple[int, int]): The initial size of the window screen,
     remains constant.
        parent (_Widget, optional): The parent widget position and size is
    relative to.

        Relative to current surface:
        relative_position (tuple[float, float]): The position of the widget
    relative to its surface.
        relative_size (tuple[float, float]): The scale of the widget relative to
    its surface.

        Remains constant, relative to initial screen size:
        relative_font_size (float, optional): The relative font size of the widget
    .
        relative_margin (float): The relative margin of the widget.
        relative_border_width (float): The relative border width of the widget.
        relative_border_radius (float): The relative border radius of the widget.

        anchor_x (str): The horizontal anchor direction ('left', 'right', 'center
    ').
        anchor_y (str): The vertical anchor direction ('top', 'bottom', 'center').
        fixed_position (tuple[int, int], optional): The fixed position of the
    widget in pixels.
        border_colour (pygame.Color): The border color of the widget.
        text_colour (pygame.Color): The text color of the widget.
        fill_colour (pygame.Color): The fill color of the widget.
        font (pygame.freetype.Font): The font used for the widget.
        """
        super().__init__()

        for required_kwarg in REQUIRED_KWARGS:
            if required_kwarg not in kwargs:
                raise KeyError(f'(_Widget.__init__) Required keyword "{
    required_kwarg}" not in base kwargs')

        self._surface = None # Set in WidgetGroup, as needs to be reassigned every
     frame
        self._raw_surface_size = DEFAULT_SURFACE_SIZE

        self._parent = kwargs.get('parent')

        self._relative_font_size = None # Set in subclass

        self._relative_position = kwargs.get('relative_position')
        self._relative_margin = theme['margin'] / self._raw_surface_size[1]
        self._relative_border_width = theme['borderWidth'] / self.
    _raw_surface_size[1]
        self._relative_border_radius = theme['borderRadius'] / self.
    _raw_surface_size[1]

        self._border_colour = pygame.Color(theme['borderPrimary'])
        self._text_colour = pygame.Color(theme['textPrimary'])
        self._fill_colour = pygame.Color(theme['fillPrimary'])
        self._font = DEFAULT_FONT

        self._anchor_x = kwargs.get('anchor_x') or 'left'
```

22

```python
        self._anchor_y = kwargs.get('anchor_y') or 'top'
        self._fixed_position = kwargs.get('fixed_position')
        scale_mode = kwargs.get('scale_mode') or 'both'

        if kwargs.get('relative_size'):
            match scale_mode:
                case 'height':
                    self._relative_size = kwargs.get('relative_size')
                case 'width':
                    self._relative_size = ((kwargs.get('relative_size')[0] * self.
surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] *
self.surface_size[0]) / self.surface_size[1])
                case 'both':
                    self._relative_size = ((kwargs.get('relative_size')[0] * self.
surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
                case _:
                    raise ValueError('(_Widget.__init__) Unknown scale mode:',
scale_mode)
        else:
            self._relative_size = (1, 1)

        if 'margin' in kwargs:
            self._relative_margin = kwargs.get('margin') / self._raw_surface_size
[1]

            if (self._relative_margin * 2) > min(self._relative_size[0], self.
_relative_size[1]):
                raise ValueError('(_Widget.__init__) Margin larger than specified
size!')

        if 'border_width' in kwargs:
            self._relative_border_width = kwargs.get('border_width') / self.
_raw_surface_size[1]

        if 'border_radius' in kwargs:
            self._relative_border_radius = kwargs.get('border_radius') / self.
_raw_surface_size[1]

        if 'border_colour' in kwargs:
            self._border_colour = pygame.Color(kwargs.get('border_colour'))

        if 'fill_colour' in kwargs:
            self._fill_colour = pygame.Color(kwargs.get('fill_colour'))

        if 'text_colour' in kwargs:
            self._text_colour = pygame.Color(kwargs.get('text_colour'))

        if 'font' in kwargs:
            self._font = kwargs.get('font')

    @property
    def surface_size(self):
        """
        Gets the size of the surface widget is drawn on.
        Can be either the window size, or another widget size if assigned to a
parent.

        Returns:
            tuple[int, int]: The size of the surface.
        """
        if self._parent:
            return self._parent.size
```

```python
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            else:
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 functionality to support widgets which rotate between text/icons.
circular.py

```
1  from data.components.circular_linked_list import CircularLinkedList
2
```

```python
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
44          Args:
45              key: The key to set as the current item.
46
47          Raises:
48              ValueError: If no nodes within the circular linked list contains the
        key as its data.
49          """
50          if self._keys_list.data_in_list(key) is False:
51              raise ValueError('(_Circular.set_to_key) Key not found:', key)
52
53          for _ in range(len(self._items_dict)):
54              if self.current_key == key:
55                  self.set_image()
56                  self.set_geometry()
57                  return
58
59              self.set_next_item()
```

26

Circular Linked List

The `CircuarLinkedList` class implements a circular doubly-linked list. Used for the internal logic of the `Circular` class.

`circular_linked_list.py`

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

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

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

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

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

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

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

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

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

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

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

27

```
56              new_node.previous = self._head.previous
57              self._head.previous.next = new_node
58              self._head.previous = new_node
59
60              self._head = new_node
61
62      def insert_at_end(self, data):
63          """
64          Inserts a node at the end of the circular linked list.
65
66          Args:
67              data: The data to insert.
68          """
69          new_node = Node(data)
70
71          if self._head is None:
72              self._head = new_node
73              new_node.next = self._head
74              new_node.previous = self._head
75          else:
76              new_node.next = self._head
77              new_node.previous = self._head.previous
78              self._head.previous.next = new_node
79              self._head.previous = new_node
80
81      def insert_at_index(self, data, index):
82          """
83          Inserts a node at a specific index in the circular linked list.
84          The head node is taken as index 0.
85
86          Args:
87              data: The data to insert.
88              index (int): The index to insert the data at.
89
90          Raises:
91              ValueError: Index is out of range.
92          """
93          if index < 0:
94              raise ValueError('Invalid index! (CircularLinkedList.insert_at_index)'
      )
95
96          if index == 0 or self._head is None:
97              self.insert_at_beginning(data)
98          else:
99              new_node = Node(data)
100             current_node = self._head
101             count = 0
102
103             while count < index - 1 and current_node.next != self._head:
104                 current_node = current_node.next
105                 count += 1
106
107             if count == (index - 1):
108                 new_node.next = current_node.next
109                 new_node.previous = current_node
110                 current_node.next = new_node
111             else:
112                 raise ValueError('Index out of range! (CircularLinkedList.
      insert_at_index)')
113
114     def delete(self, data):
115         """
```

```
116            Deletes a node with the specified data from the circular linked list.
117
118            Args:
119                data: The data to delete.
120
121            Raises:
122                ValueError: No nodes in the list contain the specified data.
123            """
124            if self._head is None:
125                return
126
127            current_node = self._head
128
129            while current_node.data != data:
130                current_node = current_node.next
131
132                if current_node == self._head:
133                    raise ValueError('Data not found in circular linked list! (
      CircularLinkedList.delete)')
134
135            if self._head.next == self._head:
136                self._head = None
137            else:
138                current_node.previous.next = current_node.next
139                current_node.next.previous = current_node.previous
140
141        def data_in_list(self, data):
142            """
143            Checks if the specified data is in the circular linked list.
144
145            Args:
146                data: The data to check.
147
148            Returns:
149                bool: True if the data is in the list, False otherwise.
150            """
151            if self._head is None:
152                return False
153
154            current_node = self._head
155            while True:
156                if current_node.data == data:
157                    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

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

- BoardThumbnailButton
- MultipleIconButton
- ReactiveIconButton
- BoardThumbnail
- ReactiveButton
- VolumeSlider
- ColourPicker
- ColourButton
- BrowserStrip
- PieceDisplay

- BrowserItem
- TextButton
- IconButton
- ScrollArea
- Chessboard
- TextInput
- Rectangle
- MoveList
- Dropdown
- Carousel

- Switch
- Timer
- Text
- Icon
- (_ColourDisplay)
- (_ColourSquare)
- (_ColourSlider)
- (_SliderThumb)
- (_Scrollbar)

CustomEvent

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

custom_event.py

```python
1  from data.constants import GameEventType, SettingsEventType, ConfigEventType,
       BrowserEventType, EditorEventType
2
3  required_args = {
4      GameEventType.BOARD_CLICK: ['coords'],
5      GameEventType.ROTATE_PIECE: ['rotation_direction'],
6      GameEventType.SET_LASER: ['laser_result'],
7      GameEventType.UPDATE_PIECES: ['move_notation'],
8      GameEventType.TIMER_END: ['active_colour'],
9      GameEventType.PIECE_DROP: ['coords', 'piece', 'colour', 'rotation', '
       remove_overlay'],
10     SettingsEventType.COLOUR_SLIDER_SLIDE: ['colour'],
11     SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK: ['colour'],
12     SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK: ['colour'],
```

```python
13        SettingsEventType.DROPDOWN_CLICK: ['selected_word'],
14        SettingsEventType.VOLUME_SLIDER_CLICK: ['volume', 'volume_type'],
15        SettingsEventType.SHADER_PICKER_CLICK: ['data'],
16        SettingsEventType.PARTICLES_CLICK: ['toggled'],
17        SettingsEventType.OPENGL_CLICK: ['toggled'],
18        ConfigEventType.TIME_TYPE: ['time'],
19        ConfigEventType.FEN_STRING_TYPE: ['time'],
20        ConfigEventType.CPU_DEPTH_CLICK: ['data'],
21        ConfigEventType.PVC_CLICK: ['data'],
22        ConfigEventType.PRESET_CLICK: ['fen_string'],
23        BrowserEventType.BROWSER_STRIP_CLICK: ['selected_index'],
24        BrowserEventType.PAGE_CLICK: ['data'],
25        EditorEventType.PICK_PIECE_CLICK: ['piece', 'active_colour'],
26        EditorEventType.ROTATE_PIECE_CLICK: ['rotation_direction'],
27    }
28
29    class CustomEvent():
30        def __init__(self, type, **kwargs):
31            self.__dict__.update(kwargs)
32            self.type = type
33
34        @classmethod
35        def create_event(event_cls, event_type, **kwargs):
36            """
37            @classmethod Factory method used to instance CustomEvent object, to check
        for required keyword arguments
38
39            Args:
40                event_cls (CustomEvent): Reference to own class.
41                event_type: The state EventType.
42
43            Raises:
44                ValueError: If required keyword argument for passed event type not
        present.
45                ValueError: If keyword argument passed is not required for passed
        event type.
46
47            Returns:
48                CustomEvent: Initialised CustomEvent instance.
49            """
50            if event_type in required_args:
51
52                for required_arg in required_args[event_type]:
53                    if required_arg not in kwargs:
54                        raise ValueError(f"Argument '{required_arg}' required for {
        event_type.name} event (GameEvent.create_event)")
55
56                for kwarg in kwargs:
57                    if kwarg not in required_args[event_type]:
58                        raise ValueError(f"Argument '{kwarg}' not included in
        required_args dictionary for event '{event_type}'! (GameEvent.create_event)")
59
60                return event_cls(event_type, **kwargs)
61
62            else:
63                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.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),
35                 border_width=0,
36                 margin=0,
37                 fit_icon=True,
38             )
39         }
40
41         super().__init__(
42             widgets_dict=widgets_dict,
43             **kwargs
44         )
```

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

```
1  from data.components.custom_event import CustomEvent
2  from data.widgets.bases.pressable import _Pressable
```

```python
from data.widgets.bases.circular import _Circular
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState

class ReactiveButton(_Pressable, _Circular, _Widget):
    def __init__(self, widgets_dict, event, center=False, **kwargs):
        # Multiple inheritance used here, to combine the functionality of multiple
super classes
        _Pressable.__init__(
            self,
            event=event,
            hover_func=lambda: self.set_to_key(WidgetState.HOVER),
            down_func=lambda: self.set_to_key(WidgetState.PRESS),
            up_func=lambda: self.set_to_key(WidgetState.BASE),
            **kwargs
        )
        # Aggregation used to cycle between external widgets
        _Circular.__init__(self, items_dict=widgets_dict)
        _Widget.__init__(self, **kwargs)

        self._center = center

        self.initialise_new_colours(self._fill_colour)

    @property
    def position(self):
        """
        Overrides position getter method, to always position icon in the center if
 self._center is True.

        Returns:
            list[int, int]: Position of widget.
        """
        position = super().position

        if self._center:
            self._size_diff = (self.size[0] - self.rect.width, self.size[1] - self
.rect.height)
            return (position[0] + self._size_diff[0] / 2, position[1] + self.
_size_diff[1] / 2)
        else:
            return position

    def set_image(self):
        """
        Sets current icon to image.
        """
        self.current_item.set_image()
        self.image = self.current_item.image

    def set_geometry(self):
        """
        Sets size and position of widget.
        """
        super().set_geometry()
        self.current_item.set_geometry()
        self.current_item.rect.topleft = self.rect.topleft

    def set_surface_size(self, new_surface_size):
        """
        Overrides base method to resize every widget state icon, not just the
current one.
```

```
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

```python
1  import pygame
2  from data.utils.widget_helpers import create_slider_gradient
3  from data.utils.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.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
```

```python
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.constants import WidgetState, CursorMode, 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.managers.cursor import cursor
10 from data.managers.theme import theme
11 from data.widgets.text import Text
12
13 logger = initialise_logger(__name__)
14
15 class TextInput(_Box, _Pressable, Text):
16     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):
17         self._cursor_index = None
18         # Multiple inheritance used here, adding the functionality of pressing,
       and custom box colours, to the text widget
19         _Box.__init__(self, box_colours=INPUT_COLOURS)
20         _Pressable.__init__(
21             self,
22             event=None,
23             hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
24             down_func=lambda: self.set_state_colour(WidgetState.PRESS),
25             up_func=lambda: self.set_state_colour(WidgetState.BASE),
26             sfx=None
27         )
28         Text.__init__(self, text="", center=False, box_colours=INPUT_COLOURS[
       WidgetState.BASE], **kwargs)
29
30         self.initialise_new_colours(self._fill_colour)
31         self.set_state_colour(WidgetState.BASE)
32
33         pygame.key.set_repeat(500, 50)
34
35         self._blinking_fps = 1000 / blinking_interval
36         self._cursor_colour = cursor_colour
37         self._cursor_colour_copy = cursor_colour
38         self._placeholder_colour = placeholder_colour
39         self._text_colour_copy = self._text_colour
40
41         self._placeholder_text = placeholder
42         self._is_placeholder = None
43         if default:
44             self._text = default
45             self.is_placeholder = False
46         else:
47             self._text = self._placeholder_text
48             self.is_placeholder = True
49
50         self._event = event
51         self._validator = validator
52         self._blinking_cooldown = 0
53
54         self._empty_cursor = pygame.Surface((0, 0), pygame.SRCALPHA)
55
56         self.resize_text()
```

37

```python
57          self.set_image ()
58          self.set_geometry ()
59
60      @property
61      # Encapsulated getter method
62      def is_placeholder ( self ):
63          return self._is_placeholder
64
65      @is_placeholder.setter
66      # Encapsulated setter method, used to replace text colour if placeholder text
        is shown
67      def is_placeholder ( self , is_true ):
68          self._is_placeholder = is_true
69
70          if is_true:
71              self._text_colour = self._placeholder_colour
72          else:
73              self._text_colour = self._text_colour_copy
74
75      @property
76      def cursor_size ( self ):
77          cursor_height = ( self.size [1] - self.border_width * 2) * 0.75
78          return ( cursor_height * 0.1 , cursor_height )
79
80      @property
81      def cursor_position ( self ):
82          current_width = ( self.margin / 2)
83          for index , metrics in enumerate ( self._font.get_metrics ( self._text , size =
        self.font_size )):
84              if index == self._cursor_index :
85                  return ( current_width - self.cursor_size [0], ( self.size [1] - self.
        cursor_size [1]) / 2)
86
87              glyph_width = metrics [4]
88              current_width += glyph_width
89          return ( current_width - self.cursor_size [0], ( self.size [1] - self.
        cursor_size [1]) / 2)
90
91      @property
92      def text ( self ):
93          if self.is_placeholder:
94              return ''
95
96          return self._text
97
98      def relative_x_to_cursor_index ( self , relative_x ):
99          """
100         Calculates cursor index using mouse position relative to the widget
        position.
101
102         Args:
103             relative_x (int): Horizontal distance of the mouse from the left side
        of the widget.
104
105         Returns:
106             int: Cursor index.
107         """
108         current_width = 0
109
110         for index , metrics in enumerate ( self._font.get_metrics ( self._text , size =
        self.font_size )):
111             glyph_width = metrics [4]
```

```python
112
113                 if current_width >= relative_x:
114                     return index
115
116                 current_width += glyph_width
117
118             return len(self._text)
119
120         def set_cursor_index(self, mouse_pos):
121             """
122             Sets cursor index based on mouse position.
123
124             Args:
125                 mouse_pos (list[int, int]): Mouse position relative to window screen.
126             """
127             if mouse_pos is None:
128                 self._cursor_index = mouse_pos
129                 return
130
131             relative_x = mouse_pos[0] - (self.margin / 2) - self.rect.left
132             relative_x = max(0, relative_x)
133             self._cursor_index = self.relative_x_to_cursor_index(relative_x)
134
135         def focus_input(self, mouse_pos):
136             """
137             Draws cursor and sets cursor index when user clicks on widget.
138
139             Args:
140                 mouse_pos (list[int, int]): Mouse position relative to window screen.
141             """
142             if self.is_placeholder:
143                 self._text = ''
144                 self.is_placeholder = False
145
146             self.set_cursor_index(mouse_pos)
147             self.set_image()
148             cursor.set_mode(CursorMode.IBEAM)
149
150         def unfocus_input(self):
151             """
152             Removes cursor when user unselects widget.
153             """
154             if self._text == '':
155                 self._text = self._placeholder_text
156                 self.is_placeholder = True
157                 self.resize_text()
158
159             self.set_cursor_index(None)
160             self.set_image()
161             cursor.set_mode(CursorMode.ARROW)
162
163         def set_text(self, new_text):
164             """
165             Called by a state object to change the widget text externally.
166
167             Args:
168                 new_text (str): New text to display.
169
170             Returns:
171                 CustomEvent: Object containing the new text to alert state of a text
       update.
172             """
```

```
173        super().set_text(new_text)
174        return CustomEvent(**vars(self._event), text=self.text)
175
176    def process_event(self, event):
177        """
178        Processes Pygame events.
179
180        Args:
181            event (pygame.Event): Event to process.
182
183        Returns:
184            CustomEvent: Object containing the new text to alert state of a text
    update.
185        """
186        previous_state = self.get_widget_state()
187        super().process_event(event)
188        current_state = self.get_widget_state()
189
190        match event.type:
191            case pygame.MOUSEMOTION:
192                if self._cursor_index is None:
193                    return
194
195                # If mouse is hovering over widget, turn mouse cursor into an I-
    beam
196                if self.rect.collidepoint(event.pos):
197                    if cursor.get_mode() != CursorMode.IBEAM:
198                        cursor.set_mode(CursorMode.IBEAM)
199                else:
200                    if cursor.get_mode() == CursorMode.IBEAM:
201                        cursor.set_mode(CursorMode.ARROW)
202
203                return
204
205            case pygame.MOUSEBUTTONUP:
206                # When user selects widget
207                if previous_state == WidgetState.PRESS:
208                    self.focus_input(event.pos)
209                # When user unselects widget
210                if current_state == WidgetState.BASE and self._cursor_index is not
    None:
211                    self.unfocus_input()
212                    return CustomEvent(**vars(self._event), text=self.text)
213
214            case pygame.KEYDOWN:
215                if self._cursor_index is None:
216                    return
217
218                # Handling Ctrl-C and Ctrl-V shortcuts
219                if event.mod & (pygame.KMOD_CTRL):
220                    if event.key == pygame.K_c:
221                        pyperclip.copy(self.text)
222                        logger.info(f'COPIED {self.text}')
223
224                    elif event.key == pygame.K_v:
225                        pasted_text = pyperclip.paste()
226                        pasted_text = ''.join(char for char in pasted_text if 32
    <= ord(char) <= 127)
227                        self._text = self._text[:self._cursor_index] + pasted_text
     + self._text[self._cursor_index:]
228                        self._cursor_index += len(pasted_text)
229
```

```
230                    elif event.key == pygame.K_BACKSPACE or event.key == pygame.
       K_DELETE:
231                        self._text = ''
232                        self._cursor_index = 0
233
234                    self.resize_text()
235                    self.set_image()
236                    self.set_geometry()
237
238                    return
239
240                match event.key:
241                    case pygame.K_BACKSPACE:
242                        if self._cursor_index > 0:
243                            self._text = self._text[:self._cursor_index - 1] +
       self._text[self._cursor_index:]
244                        self._cursor_index = max(0, self._cursor_index - 1)
245
246                    case pygame.K_RIGHT:
247                        self._cursor_index = min(len(self._text), self.
       _cursor_index + 1)
248
249                    case pygame.K_LEFT:
250                        self._cursor_index = max(0, self._cursor_index - 1)
251
252                    case pygame.K_ESCAPE:
253                        self.unfocus_input()
254                        return CustomEvent(**vars(self._event), text=self.text)
255
256                    case pygame.K_RETURN:
257                        self.unfocus_input()
258                        return CustomEvent(**vars(self._event), text=self.text)
259
260                    case _:
261                        if not event.unicode:
262                            return
263
264                        potential_text = self._text[:self._cursor_index] + event.
       unicode + self._text[self._cursor_index:]
265
266                        # Validator lambda function used to check if inputted text
        is valid before displaying
267                        # e.g. Time control input has a validator function
       checking if text represents a float
268                        if self._validator(potential_text) is False:
269                            return
270
271                        self._text = potential_text
272                        self._cursor_index += 1
273
274                self._blinking_cooldown += 1
275                animation.set_timer(500, lambda: self.subtract_blinking_cooldown
       (1))
276
277                self.resize_text()
278                self.set_image()
279                self.set_geometry()
280
281    def subtract_blinking_cooldown(self, cooldown):
282        """
283        Subtracts blinking cooldown after certain timeframe. When
       blinking_cooldown is 1, cursor is able to be drawn.
```

```
284
285          Args:
286              cooldown (float): Duration before cursor can no longer be drawn.
287          """
288          self._blinking_cooldown = self._blinking_cooldown - cooldown
289
290      def set_image(self):
291          """
292          Draws text input widget to image.
293          """
294          super().set_image()
295
296          if self._cursor_index is not None:
297              scaled_cursor = pygame.transform.scale(self._empty_cursor, self.
      cursor_size)
298              scaled_cursor.fill(self._cursor_colour)
299              self.image.blit(scaled_cursor, self.cursor_position)
300
301      def update(self):
302          """
303          Overrides based update method, to handle cursor blinking.
304          """
305          super().update()
306          # Calculate if cursor should be shown or not
307          cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
308          if cursor_frame == 1 and self._blinking_cooldown == 0:
309              self._cursor_colour = (0, 0, 0, 0)
310          else:
311              self._cursor_colour = self._cursor_colour_copy
312          self.set_image()
```

1.5   Game

1.5.1   Model

game_model.py

```
1  from data.states.game.components.fen_parser import encode_fen_string
2  from data.constants import Colour, GameEventType, EMPTY_BB
3  from data.states.game.widget_dict import GAME_WIDGETS
4  from data.states.game.cpu.cpu_thread import CPUThread
5  from data.components.custom_event import CustomEvent
6  from data.utils.bitboard_helpers import is_occupied
7  from data.states.game.components.board import Board
8  from data.utils import input_helpers as ip_helpers
9  from data.states.game.components.move import Move
10 from data.managers.logs import initialise_logger
11 from data.states.game.cpu.engines import *
12
13 logger = initialise_logger(__name__)
14
15 class GameModel:
16     def __init__(self, game_config):
17         self._listeners = {
18             'game': [],
19             'win': [],
20             'pause': [],
21         }
22         self._board = Board(fen_string=game_config['FEN_STRING'])
23
```

```python
        self.states = {
            'CPU_ENABLED': game_config['CPU_ENABLED'],
            'CPU_DEPTH': game_config['CPU_DEPTH'],
            'AWAITING_CPU': False,
            'WINNER': None,
            'PAUSED': False,
            'ACTIVE_COLOUR': game_config['COLOUR'],
            'TIME_ENABLED': game_config['TIME_ENABLED'],
            'TIME': game_config['TIME'],
            'START_FEN_STRING': game_config['FEN_STRING'],
            'MOVES': [],
            'ZOBRIST_KEYS': []
        }

        self._cpu = IDMinimaxCPU(self.states['CPU_DEPTH'], self.cpu_callback,
    verbose=False)
        self._cpu_thread = CPUThread(self._cpu)
        self._cpu_thread.start()
        self._cpu_move = None

        logger.info(f'Initialising CPU depth of {self.states['CPU_DEPTH']}')

    def register_listener(self, listener, parent_class):
        """
        Registers listener method of another MVC class.

        Args:
            listener (callable): Listener callback function.
            parent_class (str): Class name.
        """
        self._listeners[parent_class].append(listener)

    def alert_listeners(self, event):
        """
        Alerts all registered classes of an event by calling their listener
    function.

        Args:
            event (GameEventType): Event to pass as argument.

        Raises:
            Exception: If an unrecgonised event tries to be passed onto listeners.
        """
        for parent_class, listeners in self._listeners.items():
            match event.type:
                case GameEventType.UPDATE_PIECES:
                    if parent_class in 'game':
                        for listener in listeners: listener(event)

                case GameEventType.SET_LASER:
                    if parent_class == 'game':
                        for listener in listeners: listener(event)

                case GameEventType.PAUSE_CLICK:
                    if parent_class in ['pause', 'game']:
                        for listener in listeners:
                            listener(event)

                case _:
                    raise Exception('Unhandled event type (GameModel.
    alert_listeners)')
```

43

```python
83      def set_winner(self, colour=None):
84          """
85          Sets winner.
86
87          Args:
88              colour (Colour, optional): Describes winnner colour, or draw. Defaults
     to None.
89          """
90          self.states['WINNER'] = colour
91
92      def toggle_paused(self):
93          """
94          Toggles pause screen, and alerts pause view.
95          """
96          self.states['PAUSED'] = not self.states['PAUSED']
97          game_event = CustomEvent.create_event(GameEventType.PAUSE_CLICK)
98          self.alert_listeners(game_event)
99
100     def get_terminal_move(self):
101         """
102         Debugging method for inputting a move from the terminal.
103
104         Returns:
105             Move: Parsed move.
106         """
107         while True:
108             try:
109                 move_type = ip_helpers.parse_move_type(input('Input move type (m/r
     ): '))
110                 src_square = ip_helpers.parse_notation(input("From: "))
111                 dest_square = ip_helpers.parse_notation(input("To: "))
112                 rotation = ip_helpers.parse_rotation(input("Enter rotation (a/b/c/
     d): "))
113                 return Move.instance_from_notation(move_type, src_square,
     dest_square, rotation)
114             except ValueError as error:
115                 logger.warning('Input error (Board.get_move): ' + str(error))
116
117     def make_move(self, move):
118         """
119         Takes a Move object and applies it to the board.
120
121         Args:
122             move (Move): Move to apply.
123         """
124         colour = self._board.bitboards.get_colour_on(move.src)
125         piece = self._board.bitboards.get_piece_on(move.src, colour)
126         # Apply move and get results of laser trajectory
127         laser_result = self._board.apply_move(move, add_hash=True)
128
129         self.alert_listeners(CustomEvent.create_event(GameEventType.SET_LASER,
     laser_result=laser_result))
130
131         # Sets new active colour and checks for a win
132         self.states['ACTIVE_COLOUR'] = self._board.get_active_colour()
133         self.set_winner(self._board.check_win())
134
135         move_notation = move.to_notation(colour, piece, laser_result.
     hit_square_bitboard)
136
137         self.alert_listeners(CustomEvent.create_event(GameEventType.UPDATE_PIECES,
      move_notation=move_notation))
```

```python
138
139         # Adds move to move history list for review screen
140         self.states['MOVES'].append({
141             'time': {
142                 Colour.BLUE: GAME_WIDGETS['blue_timer'].get_time(),
143                 Colour.RED: GAME_WIDGETS['red_timer'].get_time()
144             },
145             'move': move_notation,
146             'laserResult': laser_result
147         })
148
149     def make_cpu_move(self):
150         """
151         Starts CPU calculations on the separate thread.
152         """
153         self.states['AWAITING_CPU'] = True
154         self._cpu_thread.start_cpu(self.get_board())
155
156     def cpu_callback(self, move):
157         """
158         Callback function passed to CPU thread. Called when CPU stops processing.
159
160         Args:
161             move (Move): Move that CPU found.
162         """
163         if self.states['WINNER'] is None:
164             # CPU move passed back to main threadby reassigning variable
165             self._cpu_move = move
166             self.states['AWAITING_CPU'] = False
167
168     def check_cpu(self):
169         """
170         Constantly checks if CPU calculations are finished, so that make_move can
    be run on the main thread.
171         """
172         if self._cpu_move is not None:
173             self.make_move(self._cpu_move)
174             self._cpu_move = None
175
176     def kill_thread(self):
177         """
178         Interrupt and kill CPU thread.
179         """
180         self._cpu_thread.kill_thread()
181         self.states['AWAITING_CPU'] = False
182
183     def is_selectable(self, bitboard):
184         """
185         Checks if square is occupied by a piece of the current active colour.
186
187         Args:
188             bitboard (int): Bitboard representing single square.
189
190         Returns:
191             bool: True if square is occupied by a piece of the current active
    colour. False if not.
192         """
193         return is_occupied(self._board.bitboards.combined_colour_bitboards[self.
    states['ACTIVE_COLOUR']], bitboard)
194
195     def get_available_moves(self, bitboard):
196         """
```

45

```
197            Gets all surrounding empty squares. Used for drawing overlay.
198
199            Args:
200                bitboard (int): Bitboard representing single center square.
201
202            Returns:
203                int: Bitboard representing all empty surrounding squares.
204            """
205            if (bitboard & self._board.get_all_active_pieces()) != EMPTY_BB:
206                return self._board.get_valid_squares(bitboard)
207
208            return EMPTY_BB
209
210        def get_piece_list(self):
211            """
212            Returns:
213                list[Piece, ...]: Array of all pieces on the board.
214            """
215            return self._board.get_piece_list()
216
217        def get_piece_info(self, bitboard):
218            """
219            Args:
220                bitboard (int): Square containing piece.
221
222            Returns:
223                tuple[Colour, Rotation, Piece]: Piece information.
224            """
225            colour = self._board.bitboards.get_colour_on(bitboard)
226            rotation = self._board.bitboards.get_rotation_on(bitboard)
227            piece = self._board.bitboards.get_piece_on(bitboard, colour)
228            return (piece, colour, rotation)
229
230        def get_fen_string(self):
231            return encode_fen_string(self._board.bitboards)
232
233        def get_board(self):
234            return self._board
```

1.5.2   View

game_view.py

```
1  import pygame
2  from data.constants import GameEventType, 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.utils.bitboard_helpers import bitboard_to_coords
9  from data.utils.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.managers.window import window
14 from data.managers.audio import audio
15 from data.assets import SFX
16
17 class GameView:
```

46

```python
18    def __init__(self, model):
19        self._model = model
20        self._hide_pieces = False
21        self._selected_coords = None
22        self._event_to_func_map = {
23            GameEventType.UPDATE_PIECES: self.handle_update_pieces,
24            GameEventType.SET_LASER: self.handle_set_laser,
25            GameEventType.PAUSE_CLICK: self.handle_pause,
26        }
27
28        # Register model event handling with process_model_event()
29        self._model.register_listener(self.process_model_event, 'game')
30
31        # Initialise WidgetGroup with map of widgets
32        self._widget_group = WidgetGroup(GAME_WIDGETS)
33        self._widget_group.handle_resize(window.size)
34        self.initialise_widgets()
35
36        self._laser_draw = LaserDraw(self.board_position, self.board_size)
37        self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
38        self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
39        self._capture_draw = CaptureDraw(self.board_position, self.board_size)
40        self._piece_group = PieceGroup()
41        self.handle_update_pieces()
42
43        self.set_status_text(StatusText.PLAYER_MOVE)
44
45    @property
46    def board_position(self):
47        return GAME_WIDGETS['chessboard'].position
48
49    @property
50    def board_size(self):
51        return GAME_WIDGETS['chessboard'].size
52
53    @property
54    def square_size(self):
55        return self.board_size[0] / 10
56
57    def initialise_widgets(self):
58        """
59        Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
60        """
61        GAME_WIDGETS['move_list'].reset_move_list()
62        GAME_WIDGETS['move_list'].kill()
63        GAME_WIDGETS['help'].kill()
64        GAME_WIDGETS['tutorial'].kill()
65
66        GAME_WIDGETS['scroll_area'].set_image()
67
68        GAME_WIDGETS['chessboard'].refresh_board()
69
70        GAME_WIDGETS['blue_piece_display'].reset_piece_list()
71        GAME_WIDGETS['red_piece_display'].reset_piece_list()
72
73    def set_status_text(self, status):
74        """
75        Sets text on status text widget.
76
77        Args:
78            status (StatusText): The game stage for which text should be displayed
     for.
```

47

```
79              """
80              match status:
81                  case StatusText.PLAYER_MOVE:
82                      GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
        ACTIVE_COLOUR'].name}'s turn to move")
83                  case StatusText.CPU_MOVE:
84                      GAME_WIDGETS['status_text'].set_text(f"CPU calculating a crazy
        move...")
85                  case StatusText.WIN:
86                      if self._model.states['WINNER'] == Miscellaneous.DRAW:
87                          GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring
        ...")
88                      else:
89                          GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
        WINNER'].name} won!")
90                  case StatusText.DRAW:
91                      GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring...")
92
93          def handle_resize(self):
94              """
95              Handles resizing of the window.
96              """
97              self._overlay_draw.handle_resize(self.board_position, self.board_size)
98              self._capture_draw.handle_resize(self.board_position, self.board_size)
99              self._piece_group.handle_resize(self.board_position, self.board_size)
100             self._laser_draw.handle_resize(self.board_position, self.board_size)
101             self._laser_draw.handle_resize(self.board_position, self.board_size)
102             self._widget_group.handle_resize(window.size)
103
104             if self._laser_draw.firing:
105                 self.update_laser_mask()
106
107         def handle_update_pieces(self, event=None):
108             """
109             Callback function to update pieces after move.
110
111             Args:
112                 event (GameEventType, optional): If updating pieces after player move,
         event contains move information. Defaults to None.
113                 toggle_timers (bool, optional): Toggle timers on and off for new
        active colour. Defaults to True.
114             """
115             piece_list = self._model.get_piece_list()
116             self._piece_group.initialise_pieces(piece_list, self.board_position, self.
        board_size)
117
118             if event:
119                 GAME_WIDGETS['move_list'].append_to_move_list(event.move_notation)
120                 GAME_WIDGETS['scroll_area'].set_image()
121                 audio.play_sfx(SFX['piece_move'])
122
123             if self._model.states['ACTIVE_COLOUR'] == Colour.BLUE:
124                 self.set_status_text(StatusText.PLAYER_MOVE)
125             elif self._model.states['CPU_ENABLED'] is False:
126                 self.set_status_text(StatusText.PLAYER_MOVE)
127             else:
128                 self.set_status_text(StatusText.CPU_MOVE)
129
130             if self._model.states['TIME_ENABLED']:
131                 self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
132                 self.toggle_timer(self._model.states['ACTIVE_COLOUR'].
        get_flipped_colour(), False)
```

```python
133
134          if self._model.states['WINNER'] is not None:
135              self.handle_game_end()
136
137      def handle_game_end(self, play_sfx=True):
138          self.toggle_timer(self._model.states['ACTIVE_COLOUR'], False)
139          self.toggle_timer(self._model.states['ACTIVE_COLOUR'].get_flipped_colour()
     , False)
140
141          if self._model.states['WINNER'] == Miscellaneous.DRAW:
142              self.set_status_text(StatusText.DRAW)
143          else:
144              self.set_status_text(StatusText.WIN)
145
146          if play_sfx:
147              audio.play_sfx(SFX['sphinx_destroy_1'])
148              audio.play_sfx(SFX['sphinx_destroy_2'])
149              audio.play_sfx(SFX['sphinx_destroy_3'])
150
151      def handle_set_laser(self, event):
152          """
153          Callback function to draw laser after move.
154
155          Args:
156              event (GameEventType): Contains laser trajectory information.
157          """
158          laser_result = event.laser_result
159
160          # If laser has hit a piece
161          if laser_result.hit_square_bitboard:
162              coords_to_remove = bitboard_to_coords(laser_result.hit_square_bitboard
     )
163              self._piece_group.remove_piece(coords_to_remove)
164
165              if laser_result.piece_colour == Colour.BLUE:
166                  GAME_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit
     )
167              elif laser_result.piece_colour == Colour.RED:
168                  GAME_WIDGETS['blue_piece_display'].add_piece(laser_result.
     piece_hit)
169
170              # Draw piece capture GFX
171              self._capture_draw.add_capture(
172                  laser_result.piece_hit,
173                  laser_result.piece_colour,
174                  laser_result.piece_rotation,
175                  coords_to_remove,
176                  laser_result.laser_path[0][0],
177                  self._model.states['ACTIVE_COLOUR']
178              )
179
180          self._laser_draw.add_laser(laser_result, self._model.states['ACTIVE_COLOUR
     '])
181          self.update_laser_mask()
182
183      def handle_pause(self, event=None):
184          """
185          Callback function for pausing timer.
186
187          Args:
188              event (None): Event argument not used.
189          """
```

```
190        is_active = not(self._model.states['PAUSED'])
191        self.toggle_timer(self._model.states['ACTIVE_COLOUR'], is_active)
192
193    def initialise_timers(self):
194        """
195        Initialises both timers with the correct amount of time and starts the
    timer for the active colour.
196        """
197        if self._model.states['TIME_ENABLED']:
198            GAME_WIDGETS['blue_timer'].set_time(self._model.states['TIME'] * 60 *
    1000)
199            GAME_WIDGETS['red_timer'].set_time(self._model.states['TIME'] * 60 *
    1000)
200        else:
201            GAME_WIDGETS['blue_timer'].kill()
202            GAME_WIDGETS['red_timer'].kill()
203
204        self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
205
206    def toggle_timer(self, colour, is_active):
207        """
208        Stops or resumes timer.
209
210        Args:
211            colour (Colour): Timer to toggle.
212            is_active (bool): Whether to pause or resume timer.
213        """
214        if colour == Colour.BLUE:
215            GAME_WIDGETS['blue_timer'].set_active(is_active)
216        elif colour == Colour.RED:
217            GAME_WIDGETS['red_timer'].set_active(is_active)
218
219    def update_laser_mask(self):
220        """
221        Uses pygame.mask to create a mask for the pieces.
222        Used for occluding the ray shader.
223        """
224        temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
225        self._piece_group.draw(temp_surface)
226        mask = pygame.mask.from_surface(temp_surface, threshold=127)
227        mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255,
    0, 0, 255))
228
229        window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
230
231    def draw(self):
232        """
233        Draws GUI and pieces onto the screen.
234        """
235        self._widget_group.update()
236        self._capture_draw.update()
237
238        self._widget_group.draw()
239        self._overlay_draw.draw(window.screen)
240
241        if self._hide_pieces is False:
242            self._piece_group.draw(window.screen)
243
244        self._laser_draw.draw(window.screen)
245        self._drag_and_drop.draw(window.screen)
246        self._capture_draw.draw(window.screen)
247
```

```
248    def process_model_event ( self , event ):
249        """
250        Registered listener function for handling GameModel events .
251        Each event is mapped to a callback function , and the appropiate one is run
       .
252
253        Args :
254            event ( GameEventType ): Game event to process .
255
256        Raises :
257            KeyError : If an unrecgonised event type is passed as the argument .
258        """
259        try :
260            self . _event_to_func_map . get ( event . type )( event )
261        except :
262            raise KeyError ( 'Event type not recognized in Game View ( GameView .
       process_model_event ): ', event . type )
263
264    def set_overlay_coords ( self , available_coords_list , selected_coord ):
265        """
266        Set board coordinates for potential moves overlay .
267
268        Args :
269            available_coords_list ( list [ tuple [ int , int ]] , ...): Array of
       coordinates
270            selected_coord ( list [ int , int ]): Coordinates of selected piece .
271        """
272        self . _selected_coords = selected_coord
273        self . _overlay_draw . set_selected_coords ( selected_coord )
274        self . _overlay_draw . set_available_coords ( available_coords_list )
275
276    def get_selected_coords ( self ):
277        return self . _selected_coords
278
279    def set_dragged_piece ( self , piece , colour , rotation ):
280        """
281        Passes information of the dragged piece to the dragging drawing class .
282
283        Args :
284            piece ( Piece ): Piece type of dragged piece .
285            colour ( Colour ): Colour of dragged piece .
286            rotation ( Rotation ): Rotation of dragged piece .
287        """
288        self . _drag_and_drop . set_dragged_piece ( piece , colour , rotation )
289
290    def remove_dragged_piece ( self ):
291        """
292        Stops drawing dragged piece when user lets go of piece .
293        """
294        self . _drag_and_drop . remove_dragged_piece ()
295
296    def convert_mouse_pos ( self , event ):
297        """
298        Passes information of what mouse cursor is interacting with to a
       GameController object .
299
300        Args :
301            event ( pygame . Event ): Mouse event to process .
302
303        Returns :
304            CustomEvent | None : Contains information what mouse is doing .
305        """
```

```
306          clicked_coords = screen_pos_to_coords(event.pos, self.board_position, self
      .board_size)
307
308          if event.type == pygame.MOUSEBUTTONDOWN:
309              if clicked_coords:
310                  return CustomEvent.create_event(GameEventType.BOARD_CLICK, coords=
      clicked_coords)
311
312              else:
313                  return None
314
315          elif event.type == pygame.MOUSEBUTTONUP:
316              if self._drag_and_drop.dragged_sprite:
317                  piece, colour, rotation = self._drag_and_drop.get_dragged_info()
318                  piece_dragged = self._drag_and_drop.remove_dragged_piece()
319                  return CustomEvent.create_event(GameEventType.PIECE_DROP, coords=
      clicked_coords, piece=piece, colour=colour, rotation=rotation, remove_overlay=
      piece_dragged)
320
321      def add_help_screen(self):
322          """
323          Draw help overlay when player clicks on the help button.
324          """
325          self._widget_group.add(GAME_WIDGETS['help'])
326          self._widget_group.handle_resize(window.size)
327
328      def add_tutorial_screen(self):
329          """
330          Draw tutorial overlay when player clicks on the tutorial button.
331          """
332          self._widget_group.add(GAME_WIDGETS['tutorial'])
333          self._widget_group.handle_resize(window.size)
334          self._hide_pieces = True
335
336      def remove_help_screen(self):
337          GAME_WIDGETS['help'].kill()
338
339      def remove_tutorial_screen(self):
340          GAME_WIDGETS['tutorial'].kill()
341          self._hide_pieces = False
342
343      def process_widget_event(self, event):
344          """
345          Passes Pygame event to WidgetGroup to allow individual widgets to process
      events.
346
347          Args:
348              event (pygame.Event): Event to process.
349
350          Returns:
351              CustomEvent | None: A widget event.
352          """
353          return self._widget_group.process_event(event)
```

1.5.3   Controller

game_controller.py

```
1 import pygame
2 from data.constants import GameEventType, MoveType, StatusText, Miscellaneous
3 from data.utils import bitboard_helpers as bb_helpers
```

```python
4  from data.states.game.components.move import Move
5  from data.managers.logs import initialise_logger
6
7  logger = initialise_logger(__name__)
8
9  class GameController:
10     def __init__(self, model, view, win_view, pause_view, to_menu, to_review,
       to_new_game):
11         self._model = model
12         self._view = view
13         self._win_view = win_view
14         self._pause_view = pause_view
15
16         self._to_menu = to_menu
17         self._to_review = to_review
18         self._to_new_game = to_new_game
19
20         self._view.initialise_timers()
21         self._win_view.set_win_type('CAPTURE')
22
23     def cleanup(self, next):
24         """
25         Handles game quit, either leaving to main menu or restarting a new game.
26
27         Args:
28             next (str): New state to switch to.
29         """
30         self._model.kill_thread()
31
32         if next == 'menu':
33             self._to_menu()
34         elif next == 'game':
35             self._to_new_game()
36         elif next == 'review':
37             self._to_review()
38
39     def make_move(self, move):
40         """
41         Handles player move.
42
43         Args:
44             move (Move): Move to make.
45         """
46         self._model.make_move(move)
47         self._view.set_overlay_coords([], None)
48
49         if self._model.states['CPU_ENABLED']:
50             self._model.make_cpu_move()
51
52     def handle_pause_event(self, event):
53         """
54         Processes events when game is paused.
55
56         Args:
57             event (GameEventType): Event to process.
58
59         Raises:
60             Exception: If event type is unrecognised.
61         """
62         game_event = self._pause_view.convert_mouse_pos(event)
63
64         if game_event is None:
```

```python
65                 return
66
67         match game_event.type:
68             case GameEventType.PAUSE_CLICK:
69                 self._model.toggle_paused()
70
71             case GameEventType.MENU_CLICK:
72                 self.cleanup('menu')
73
74             case _:
75                 raise Exception('Unhandled event type (GameController.handle_event
       )')
76
77     def handle_winner_event(self, event):
78         """
79         Processes events when game is over.
80
81         Args:
82             event (GameEventType): Event to process.
83
84         Raises:
85             Exception: If event type is unrecognised.
86         """
87         game_event = self._win_view.convert_mouse_pos(event)
88
89         if game_event is None:
90             return
91
92         match game_event.type:
93             case GameEventType.MENU_CLICK:
94                 self.cleanup('menu')
95                 return
96
97             case GameEventType.GAME_CLICK:
98                 self.cleanup('game')
99                 return
100
101            case GameEventType.REVIEW_CLICK:
102                self.cleanup('review')
103
104            case _:
105                raise Exception('Unhandled event type (GameController.handle_event
       )')
106
107    def handle_game_widget_event(self, event):
108        """
109        Processes events for game GUI widgets.
110
111        Args:
112            event (GameEventType): Event to process.
113
114        Raises:
115            Exception: If event type is unrecognised.
116
117        Returns:
118            CustomEvent | None: A widget event.
119        """
120        widget_event = self._view.process_widget_event(event)
121
122        if widget_event is None:
123            return None
124
```

```python
            match widget_event.type:
                case GameEventType.ROTATE_PIECE:
                    src_coords = self._view.get_selected_coords()

                    if src_coords is None:
                        logger.info('None square selected')
                        return

                    move = Move.instance_from_coords(MoveType.ROTATE, src_coords,
    src_coords, rotation_direction=widget_event.rotation_direction)
                    self.make_move(move)

                case GameEventType.RESIGN_CLICK:
                    self._model.set_winner(self._model.states['ACTIVE_COLOUR'].
    get_flipped_colour())
                    self._view.handle_game_end(play_sfx=False)
                    self._win_view.set_win_type('RESIGN')

                case GameEventType.DRAW_CLICK:
                    self._model.set_winner(Miscellaneous.DRAW)
                    self._view.handle_game_end(play_sfx=False)
                    self._win_view.set_win_type('DRAW')

                case GameEventType.TIMER_END:
                    if self._model.states['TIME_ENABLED']:
                        self._model.set_winner(widget_event.active_colour.
    get_flipped_colour())
                        self._win_view.set_win_type('TIME')
                        self._view.handle_game_end(play_sfx=False)

                case GameEventType.MENU_CLICK:
                    self.cleanup('menu')

                case GameEventType.HELP_CLICK:
                    self._view.add_help_screen()

                case GameEventType.TUTORIAL_CLICK:
                    self._view.add_tutorial_screen()

                case _:
                    raise Exception('Unhandled event type (GameController.handle_event
    )')

        return widget_event.type

    def check_cpu(self):
        """
        Checks if CPU calculations are finished every frame.
        """
        if self._model.states['CPU_ENABLED'] and self._model.states['AWAITING_CPU'
    ] is False:
            self._model.check_cpu()

    def handle_game_event(self, event):
        """
        Processes Pygame events for main game.

        Args:
            event (pygame.Event): If event type is unrecognised.

        Raises:
            Exception: If event type is unrecognised.
```

```
182          """
183          # Pass event for widgets to process
184          widget_event = self.handle_game_widget_event(event)
185
186          if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
      KEYDOWN]:
187              if event.type != pygame.KEYDOWN:
188                  game_event = self._view.convert_mouse_pos(event)
189              else:
190                  game_event = None
191
192              if game_event is None:
193                  if widget_event is None:
194                      if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
195                          # If user releases mouse click not on a widget
196                          self._view.remove_help_screen()
197                          self._view.remove_tutorial_screen()
198                      if event.type == pygame.MOUSEBUTTONUP:
199                          # If user releases mouse click on neither a widget or
      board
200                          self._view.set_overlay_coords(None, None)
201
202                  return
203
204              match game_event.type:
205                  case GameEventType.BOARD_CLICK:
206                      if self._model.states['AWAITING_CPU']:
207                          return
208
209                      clicked_coords = game_event.coords
210                      clicked_bitboard = bb_helpers.coords_to_bitboard(
      clicked_coords)
211                      selected_coords = self._view.get_selected_coords()
212
213                      if selected_coords:
214                          if clicked_coords == selected_coords:
215                              # If clicking on an already selected square, start
      dragging piece on that square
216                              self._view.set_dragged_piece(*self._model.
      get_piece_info(clicked_bitboard))
217                              return
218
219                          selected_bitboard = bb_helpers.coords_to_bitboard(
      selected_coords)
220                          available_bitboard = self._model.get_available_moves(
      selected_bitboard)
221
222                          if bb_helpers.is_occupied(clicked_bitboard,
      available_bitboard):
223                              # If the newly clicked square is not the same as the
      old one, and is an empty surrounding square, make a move
224                              move = Move.instance_from_coords(MoveType.MOVE,
      selected_coords, clicked_coords)
225                              self.make_move(move)
226                          else:
227                              # If the newly clicked square is not the same as the
      old one, but is an invalid square, unselect the currently selected square
228                              self._view.set_overlay_coords(None, None)
229
230                      # Select hovered square if it is same as active colour
231                      elif self._model.is_selectable(clicked_bitboard):
```

56

```
232                            available_bitboard = self._model.get_available_moves(
        clicked_bitboard)
233                            self._view.set_overlay_coords(bb_helpers.
        bitboard_to_coords_list(available_bitboard), clicked_coords)
234                            self._view.set_dragged_piece(*self._model.get_piece_info(
        clicked_bitboard))
235
236                    case GameEventType.PIECE_DROP:
237                        hovered_coords = game_event.coords
238
239                        # if piece is dropped onto the board
240                        if hovered_coords:
241                            hovered_bitboard = bb_helpers.coords_to_bitboard(
        hovered_coords)
242                            selected_coords = self._view.get_selected_coords()
243                            selected_bitboard = bb_helpers.coords_to_bitboard(
        selected_coords)
244                            available_bitboard = self._model.get_available_moves(
        selected_bitboard)
245
246                            if bb_helpers.is_occupied(hovered_bitboard,
        available_bitboard):
247                                # Make a move if mouse is hovered over an empty
        surrounding square
248                                move = Move.instance_from_coords(MoveType.MOVE,
        selected_coords, hovered_coords)
249                                self.make_move(move)
250
251                        if game_event.remove_overlay:
252                            self._view.set_overlay_coords(None, None)
253
254                        self._view.remove_dragged_piece()
255
256                    case _:
257                        raise Exception('Unhandled event type (GameController.
        handle_event)', game_event.type)
258
259    def handle_event(self, event):
260        """
261        Passe a Pygame event to the correct handling function according to the
        game state.
262
263        Args:
264            event (pygame.Event): Event to process.
265        """
266        if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
        MOUSEMOTION, pygame.KEYDOWN]:
267            if self._model.states['PAUSED']:
268                self.handle_pause_event(event)
269            elif self._model.states['WINNER'] is not None:
270                self.handle_winner_event(event)
271            else:
272                self.handle_game_event(event)
273
274        if event.type == pygame.KEYDOWN:
275            if event.key == pygame.K_ESCAPE:
276                self._model.toggle_paused()
277            elif event.key == pygame.K_l:
278                logger.info('\nSTOPPING CPU')
279                self._model._cpu_thread.stop_cpu() #temp
```

1.5.4 Board

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

board.py

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

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

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

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

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

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

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

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

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

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

```python
53
54     def get_active_colour(self):
55         """
56         Gets the active colour.
57
58         Returns:
59             Colour: The active colour.
60         """
61         return self.bitboards.active_colour
62
63     def to_hash(self):
64         """
65         Gets the hash of the current board state.
66
67         Returns:
68             int: A Zobrist hash.
69         """
70         return self.bitboards.get_hash()
71
72     def check_win(self):
73         """
74         Checks for a Pharoah capture or threefold-repetition.
75
76         Returns:
77             Colour | Miscellaneous: The winning colour, or Miscellaneous.DRAW.
78         """
79         for colour in Colour:
80             if self.bitboards.get_piece_bitboard(Piece.PHAROAH, colour) ==
       EMPTY_BB:
81                 return colour.get_flipped_colour()
82
83         if self.hash_list.count(self.hash_list[-1]) >= 3:
84             return Miscellaneous.DRAW
85
86         return None
87
88     def apply_move(self, move, fire_laser=True, add_hash=False):
89         """
90         Applies a move to the board.
91
92         Args:
93             move (Move): The move to apply.
94             fire_laser (bool): Whether to fire the laser after the move.
95             add_hash (bool): Whether to add the board state hash to the hash list.
96
97         Returns:
98             Laser: The laser trajectory result.
99         """
100        piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
       active_colour)
101
102        if piece_symbol is None:
103            raise ValueError('Invalid move - no piece found on source square')
104        elif piece_symbol == Piece.SPHINX:
105            raise ValueError('Invalid move - sphinx piece is immovable')
106
107        if move.move_type == MoveType.MOVE:
108            possible_moves = self.get_valid_squares(move.src)
109            if bb_helpers.is_occupied(move.dest, possible_moves) is False:
110                raise ValueError('Invalid move - destination square is occupied')
111
112            piece_rotation = self.bitboards.get_rotation_on(move.src)
```

```python
113
114            self.bitboards.update_move(move.src, move.dest)
115            self.bitboards.update_rotation(move.src, move.dest, piece_rotation)
116
117        elif move.move_type == MoveType.ROTATE:
118            piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
    active_colour)
119            piece_rotation = self.bitboards.get_rotation_on(move.src)
120
121            if move.rotation_direction == RotationDirection.CLOCKWISE:
122                new_rotation = piece_rotation.get_clockwise()
123            elif move.rotation_direction == RotationDirection.ANTICLOCKWISE:
124                new_rotation = piece_rotation.get_anticlockwise()
125
126            self.bitboards.update_rotation(move.src, move.src, new_rotation)
127
128        laser = None
129        if fire_laser:
130            laser = self.fire_laser(add_hash)
131
132        if add_hash:
133            self.hash_list.append(self.bitboards.get_hash())
134
135        self.bitboards.flip_colour()
136
137        return laser
138
139    def undo_move(self, move, laser_result):
140        """
141        Undoes a move on the board.
142
143        Args:
144            move (Move): The move to undo.
145            laser_result (Laser): The laser trajectory result.
146        """
147        self.bitboards.flip_colour()
148
149        if laser_result.hit_square_bitboard:
150            # Get info of destroyed piece, and add it to the board again
151            src = laser_result.hit_square_bitboard
152            piece = laser_result.piece_hit
153            colour = laser_result.piece_colour
154            rotation = laser_result.piece_rotation
155
156            self.bitboards.set_square(src, piece, colour)
157            self.bitboards.clear_rotation(src)
158            self.bitboards.set_rotation(src, rotation)
159
160        # Create new Move object that is the inverse of the passed move
161        if move.move_type == MoveType.MOVE:
162            reversed_move = Move.instance_from_bitboards(MoveType.MOVE, move.dest,
     move.src)
163        elif move.move_type == MoveType.ROTATE:
164            reversed_move = Move.instance_from_bitboards(MoveType.ROTATE, move.src
    , move.src, move.rotation_direction.get_opposite())
165
166        self.apply_move(reversed_move, fire_laser=False)
167        self.bitboards.flip_colour()
168
169    def remove_piece(self, square_bitboard):
170        """
171        Removes a piece from a given square.
```

```
172
173        Args:
174            square_bitboard (int): The bitboard representation of the square.
175        """
176        self.bitboards.clear_square(square_bitboard, Colour.BLUE)
177        self.bitboards.clear_square(square_bitboard, Colour.RED)
178        self.bitboards.clear_rotation(square_bitboard)
179
180    def get_valid_squares(self, src_bitboard, colour=None):
181        """
182        Gets valid squares for a piece to move to.
183
184        Args:
185            src_bitboard (int): The bitboard representation of the source square.
186            colour (Colour, optional): The active colour of the piece.
187
188        Returns:
189            int: The bitboard representation of valid squares.
190        """
191        target_top_left = (src_bitboard & A_FILE_MASK & EIGHT_RANK_MASK) << 9
192        target_top_middle = (src_bitboard & EIGHT_RANK_MASK) << 10
193        target_top_right = (src_bitboard & J_FILE_MASK & EIGHT_RANK_MASK) << 11
194        target_middle_right = (src_bitboard & J_FILE_MASK) << 1
195
196        target_bottom_right = (src_bitboard & J_FILE_MASK & ONE_RANK_MASK) >> 9
197        target_bottom_middle = (src_bitboard & ONE_RANK_MASK) >> 10
198        target_bottom_left = (src_bitboard & A_FILE_MASK & ONE_RANK_MASK)>> 11
199        target_middle_left = (src_bitboard & A_FILE_MASK) >> 1
200
201        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
202
203        if colour is not None:
204            valid_possible_moves = possible_moves & ~self.bitboards.
        combined_colour_bitboards[colour]
205        else:
206            valid_possible_moves = possible_moves & ~self.bitboards.
        combined_all_bitboard
207
208        return valid_possible_moves
209
210    def get_mobility(self, colour):
211        """
212        Gets all valid squares for a given colour.
213
214        Args:
215            colour (Colour): The colour of the pieces.
216
217        Returns:
218            int: The bitboard representation of all valid squares.
219        """
220        active_pieces = self.get_all_active_pieces(colour)
221        possible_moves = 0
222
223        for square in bb_helpers.occupied_squares(active_pieces):
224            possible_moves += bb_helpers.pop_count(self.get_valid_squares(square))
225
226        return possible_moves
227
228    def get_all_active_pieces(self, colour=None):
229        """
```

```
230             Gets all active pieces for the current player.
231
232             Args:
233                 colour (Colour): Active colour of pieces to retrieve. Defaults to None
       .
234
235             Returns:
236                 int: The bitboard representation of all active pieces.
237             """
238             if colour is None:
239                 colour = self.bitboards.active_colour
240
241             active_pieces = self.bitboards.combined_colour_bitboards[colour]
242             sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
243             return active_pieces ^ sphinx_bitboard
244
245         def fire_laser(self, remove_hash):
246             """
247             Fires the laser and removes hit pieces.
248
249             Args:
250                 remove_hash (bool): Whether to clear the hash list if a piece is hit.
251
252             Returns:
253                 Laser: The result of firing the laser.
254             """
255             laser = Laser(self.bitboards)
256
257             if laser.hit_square_bitboard:
258                 self.remove_piece(laser.hit_square_bitboard)
259
260                 if remove_hash:
261                     self.hash_list = [] # Remove all hashes for threefold repetition,
       as the position is impossible to be repeated after a piece is removed
262             return laser
263
264         def generate_square_moves(self, src):
265             """
266             Generates all valid moves for a piece on a given square.
267
268             Args:
269                 src (int): The bitboard representation of the source square.
270
271             Yields:
272                 Move: A valid move for the piece.
273             """
274             for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
275                 yield Move(MoveType.MOVE, src, dest)
276
277         def generate_all_moves(self, colour):
278             """
279             Generates all valid moves for a given colour.
280
281             Args:
282                 colour (Colour): The colour of the pieces.
283
284             Yields:
285                 Move: A valid move for the active colour.
286             """
287             sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
288             # Remove source squares for Sphinx pieces, as they cannot be moved
289             sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour]
```

```
      ^ sphinx_bitboard
290
291       for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
292           # Generate movement moves
293           yield from self.generate_square_moves(square)
294
295           # Generate rotational moves
296           for rotation_direction in RotationDirection:
297               yield Move(MoveType.ROTATE, square, rotation_direction=
      rotation_direction)
```

1.5.5 Bitboards

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

```
40
41                 if piece is not None:
42                     characters += f'{piece.upper() if colour == Colour.BLUE
     else piece}  '
43                 else:
44                     characters += '.  '
45
46             characters += '\n\n'
47
48         return characters
49
50     def get_rotation_string(self):
51         """
52         Returns a string representation of the board rotations.
53
54         Returns:
55             str: Board formatted with only rotations shown.
56         """
57         characters = ''
58         for rank in reversed(Rank):
59
60             for file in File:
61                 mask = 1 << (rank * 10 + file)
62                 rotation = self.get_rotation_on(mask)
63                 has_piece = bb_helpers.is_occupied(self.combined_all_bitboard,
     mask)
64
65                 if has_piece:
66                     characters += f'{rotation.upper()}  '
67                 else:
68                     characters += '.  '
69
70             characters += '\n\n'
71
72         return characters
73
74     def initialise_hash(self):
75         """
76         Initialises the Zobrist hash for the current board state.
77         """
78         for piece in Piece:
79             for colour in Colour:
80                 piece_bitboard = self.get_piece_bitboard(piece, colour)
81
82                 for occupied_bitboard in bb_helpers.occupied_squares(
     piece_bitboard):
83                     self._hasher.apply_piece_hash(occupied_bitboard, piece, colour
     )
84
85         for bitboard in bb_helpers.loop_all_squares():
86             rotation = self.get_rotation_on(bitboard)
87             self._hasher.apply_rotation_hash(bitboard, rotation)
88
89         if self.active_colour == Colour.RED:
90             self._hasher.apply_red_move_hash()
91
92     def flip_colour(self):
93         """
94         Flips the active colour and updates the Zobrist hash.
95         """
96         self.active_colour = self.active_colour.get_flipped_colour()
97
```

64

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

65

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

```python
210
211        self._hasher.apply_piece_hash(bitboard, piece, colour)
212
213    def get_piece_bitboard(self, piece, colour):
214        """
215        Gets the bitboard for a piece type for a given colour.
216
217        Args:
218            piece (Piece): The piece bitboard to get.
219            colour (Colour): The colour of the piece.
220
221        Returns:
222            int: The bitboard representation for all squares occupied by that
    piece type.
223        """
224        return self.piece_bitboards[colour][piece]
225
226    def get_piece_on(self, target_bitboard, colour):
227        """
228        Gets the piece on a given square for a given colour.
229
230        Args:
231            target_bitboard (int): The bitboard representation of the square.
232            colour (Colour): The colour of the piece.
233
234        Returns:
235            Piece: The piece on the square, or None if square is empty.
236        """
237        if not (bb_helpers.is_occupied(self.combined_colour_bitboards[colour],
    target_bitboard)):
238            return None
239
240        return next(
241            (piece for piece in Piece if
242                bb_helpers.is_occupied(self.get_piece_bitboard(piece, colour),
    target_bitboard)),
243            None)
244
245    def get_rotation_on(self, target_bitboard):
246        """
247        Gets the rotation on a given square.
248
249        Args:
250            target_bitboard (int): The bitboard representation of the square.
251
252        Returns:
253            Rotation: The rotation on the square.
254        """
255        rotationBits = [bb_helpers.is_occupied(self.rotation_bitboards[
    RotationIndex.SECONDBIT], target_bitboard), bb_helpers.is_occupied(self.
    rotation_bitboards[RotationIndex.FIRSTBIT], target_bitboard)]
256
257        match rotationBits:
258            case [False, False]:
259                return Rotation.UP
260            case [False, True]:
261                return Rotation.RIGHT
262            case [True, False]:
263                return Rotation.DOWN
264            case [True, True]:
265                return Rotation.LEFT
266
```

```python
267    def get_colour_on ( self , target_bitboard ):
268        """
269        Gets the colour of the piece on a given square .
270
271        Args :
272            target_bitboard ( int ): The bitboard representation of the square .
273
274        Returns :
275            Colour : The colour of the piece on the square .
276        """
277        for piece in Piece :
278            if self . get_piece_bitboard ( piece , Colour . BLUE ) & target_bitboard !=
    EMPTY_BB :
279                return Colour . BLUE
280            elif self . get_piece_bitboard ( piece , Colour . RED ) & target_bitboard !=
    EMPTY_BB :
281                return Colour . RED
282
283    def get_piece_count ( self , piece , colour ):
284        """
285        Gets the count of a given piece type and colour .
286
287        Args :
288            piece ( Piece ): The piece to count .
289            colour ( Colour ): The colour of the piece .
290
291        Returns :
292            int : The number of that piece of that colour on the board .
293        """
294        return bb_helpers . pop_count ( self . get_piece_bitboard ( piece , colour ))
295
296    def get_hash ( self ):
297        """
298        Gets the Zobrist hash of the current board state .
299
300        Returns :
301            int : The Zobrist hash .
302        """
303        return self . _hasher . hash
304
305    def convert_to_piece_list ( self ):
306        """
307        Converts all bitboards to a list of pieces .
308
309        Returns :
310            list : Board represented as a 2D list of Piece and Rotation objects .
311        """
312        piece_list = []
313
314        for i in range (80):
315            if x := self . get_piece_on (1 << i , Colour . BLUE ):
316                rotation = self . get_rotation_on (1 << i )
317                piece_list . append (( x . upper (), rotation ))
318            elif y := self . get_piece_on (1 << i , Colour . RED ):
319                rotation = self . get_rotation_on (1 << i )
320                piece_list . append (( y , rotation ))
321            else :
322                piece_list . append ( None )
323
324        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.

1.6.1 Minimax

minimax.py

```python
1  from data.states.game.cpu.base import BaseCPU
2  from data.constants import Score, Colour
3  from random import choice
4
5  class MinimaxCPU(BaseCPU):
6      def __init__(self, max_depth, callback, verbose=False):
7          super().__init__(callback, verbose)
8          self._max_depth = max_depth
9
10     def find_move(self, board, stop_event):
11         """
12         Finds the best move for the current board state.
13
14         Args:
15             board (Board): The current board state.
16             stop_event (threading.Event): Event used to kill search from an
    external class.
17         """
18         self.initialise_stats()
19         best_score, best_move = self.search(board, self._max_depth, stop_event)
20
21         if self._verbose:
22             self.print_stats(best_score, best_move)
23
24         self._callback(best_move)
25
26     def search(self, board, depth, stop_event):
27         """
28         Recursively DFS through minimax tree with evaluation score.
29
30         Args:
31             board (Board): The current board state.
32             depth (int): The current search depth.
33             stop_event (threading.Event): Event used to kill search from an
    external class.
34         Returns:
35             tuple[int, Move]: The best score and the best move found.
36         """
37         if (base_case := super().search(board, depth, stop_event)):
38             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

alpha_beta.py

```
1
2  from data.states.game.cpu.move_orderer import MoveOrderer
3  from data.states.game.cpu.base import BaseCPU
4  from data.constants import Score, Colour
5  from random import choice
```

```
 6
 7   class ABMinimaxCPU ( BaseCPU ):
 8       def __init__ ( self , max_depth , callback , verbose = True ):
 9           super () . __init__ ( callback , verbose )
10           self . _max_depth = max_depth
11           self . _orderer = MoveOrderer ()
12
13       def initialise_stats ( self ):
14           """
15           Initialises the number of prunes to the statistics dictionary to be logged
       .
16           """
17           super () . initialise_stats ()
18           self . _stats [ 'beta_prunes' ] = 0
19           self . _stats [ 'alpha_prunes' ] = 0
20
21       def find_move ( self , board , stop_event ):
22           """
23           Finds the best move for the current board state .
24
25           Args :
26               board ( Board ): The current board state .
27               stop_event ( threading . Event ): Event used to kill search from an
       external class .
28           """
29           self . initialise_stats ()
30           best_score , best_move = self . search ( board , self . _max_depth , - Score .
       INFINITE , Score . INFINITE , stop_event )
31
32           if self . _verbose :
33               self . print_stats ( best_score , best_move )
34
35           self . _callback ( best_move )
36
37       def search ( self , board , depth , alpha , beta , stop_event , hint = None ,
       laser_coords = None ):
38           """
39           Recursively DFS through minimax tree while pruning branches using the
       alpha and beta bounds .
40
41           Args :
42               board ( Board ): The current board state .
43               depth ( int ): The current search depth .
44               alpha ( int ): The upper bound value .
45               beta ( int ): The lower bound value .
46               stop_event ( threading . Event ): Event used to kill search from an
       external class .
47
48           Returns :
49               tuple [ int , Move ]: The best score and the best move found .
50           """
51           if ( base_case := super () . search ( board , depth , stop_event )):
52               return base_case
53
54           best_move = None
55
56           # Blue is the maximising player
57           if board . get_active_colour () == Colour . BLUE :
58               max_score = - Score . INFINITE
59
60               for move in self . _orderer . get_moves ( board , hint = hint , laser_coords =
       laser_coords ):
```

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

1.6.3   Transposition Table

For adding transposition table functionality to my other engine classes, 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.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
2  from data.states.game.cpu.transposition_table import TranspositionTable
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
```

1.6.4   Evaluator

evaluator.py

```
1  from data.utils.bitboard_helpers import pop_count, occupied_squares,
       bitboard_to_index
2  from data.states.game.components.psqt import PSQT, FLIP
3  from data.managers.logs import initialise_logger
4  from data.constants import Colour, Piece, Score
5
6  logger = initialise_logger(__name__)
```

```python
7
8  class Evaluator:
9      def __init__(self, verbose=True):
10         self._verbose = verbose
11
12     def evaluate(self, board, absolute=False):
13         """
14         Evaluates and returns a numerical score for the board state.
15
16         Args:
17             board (Board): The current board state.
18             absolute (bool): Whether to always return the absolute score from the
    active colour's perspective (for NegaMax).
19
20         Returns:
21             int: Score representing advantage/disadvantage for the player.
22         """
23         blue_score = (
24             self.evaluate_material(board, Colour.BLUE),
25             self.evaluate_position(board, Colour.BLUE),
26             self.evaluate_mobility(board, Colour.BLUE),
27             self.evaluate_pharoah_safety(board, Colour.BLUE)
28         )
29
30         red_score = (
31             self.evaluate_material(board, Colour.RED),
32             self.evaluate_position(board, Colour.RED),
33             self.evaluate_mobility(board, Colour.RED),
34             self.evaluate_pharoah_safety(board, Colour.RED)
35         )
36
37         if self._verbose:
38             logger.info(f'Material: {blue_score[0]} | {red_score[0]}')
39             logger.info(f'Position: {blue_score[1]} | {red_score[1]}')
40             logger.info(f'Mobility: {blue_score[2]} | {red_score[2]}')
41             logger.info(f'Safety: {blue_score[3]} | {red_score[3]}')
42             logger.info(f'Overall score: {sum(blue_score) - sum(red_score)}')
43
44         if absolute and board.get_active_colour() == Colour.RED:
45             return sum(red_score) - sum(blue_score)
46         else:
47             return sum(blue_score) - sum(red_score)
48
49     def evaluate_material(self, board, colour):
50         """
51         Evaluates the material score for a given colour.
52
53         Args:
54             board (Board): The current board state.
55             colour (Colour): The colour to evaluate.
56
57         Returns:
58             int: Sum of all piece scores.
59         """
60         return (
61             Score.SPHINX * board.bitboards.get_piece_count(Piece.SPHINX, colour) +
62             Score.PYRAMID * board.bitboards.get_piece_count(Piece.PYRAMID, colour)
    +
63             Score.ANUBIS * board.bitboards.get_piece_count(Piece.ANUBIS, colour) +
64             Score.SCARAB * board.bitboards.get_piece_count(Piece.SCARAB, colour)
65         )
66
```

```python
67     def evaluate_position(self, board, colour):
68         """
69         Evaluates the positional score for a given colour.
70
71         Args:
72             board (Board): The current board state.
73             colour (Colour): The colour to evaluate.
74
75         Returns:
76             int: Score representing positional advantage/disadvantage.
77         """
78         score = 0
79
80         for piece in Piece:
81             if piece == Piece.SPHINX:
82                 continue
83
84             piece_bitboard = board.bitboards.get_piece_bitboard(piece, colour)
85
86             for bitboard in occupied_squares(piece_bitboard):
87                 index = bitboard_to_index(bitboard)
88                 # Flip PSQT if using from blue player's perspective
89                 index = FLIP[index] if colour == Colour.BLUE else index
90
91                 score += PSQT[piece][index] * Score.POSITION
92
93         return score
94
95     def evaluate_mobility(self, board, colour):
96         """
97         Evaluates the mobility score for a given colour.
98
99         Args:
100            board (Board): The current board state.
101            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_pharoah_safety(self, board, colour):
110        """
111        Evaluates the safety of the Pharoah 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 Pharoah.
119        """
120        pharoah_bitboard = board.bitboards.get_piece_bitboard(Piece.PHAROAH,
       colour)
121
122        if pharoah_bitboard:
123            pharoah_available_moves = pop_count(board.get_valid_squares(
       pharoah_bitboard, colour))
124            return (8 - pharoah_available_moves) * Score.PHAROAH_SAFETY
125        else:
126            return 0
```

1.6.5 Multithreading

A `CPUThread` is initialised with a CPU engine at the start of the game state, and run whenever it is the CPU's turn to move.

cpu_thread.py

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

logger = initialise_logger(__name__)

class CPUThread(threading.Thread):
    def __init__(self, cpu, verbose=False):
        super().__init__()
        self._stop_event = threading.Event()
        self._running = True
        self._verbose = verbose
        self.daemon = True

        self._board = None
        self._cpu = cpu

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

    def stop_cpu(self):
        """
        Kills the CPU's move search.
        """
        self._stop_event.set()
        self._board = None

    def start_cpu(self, board):
        """
        Starts the CPU's move search.

        Args:
            board (Board): The current board state.
        """
        self._stop_event.clear()
        self._board = board

    def run(self):
        """
        Periodically checks if the board variable is set.
        If it is, then starts CPU search.
        """
        while self._running:
            if self._board and self._cpu:
                self._cpu.find_move(self._board, self._stop_event)
                self.stop_cpu()
            else:
                time.sleep(1)
                if self._verbose:
                    logger.debug(f'(CPUThread.run) Thread {threading.get_native_id
    ()} idling...')
```

1.6.6 Zobrist Hashing

zobrist_hasher.py

```python
from random import randint
from data.utils.bitboard_helpers import bitboard_to_index
from data.constants import Piece, Colour, Rotation

# Initialise random values for each piece type on every square
# (5 x 2 colours) pieces + 4 rotations, for 80 squares
zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)]
# Hash for when the red player's move
red_move_hash = randint(0, 2 ** 64)

# Maps piece to the correct random value
piece_lookup = {
    Colour.BLUE: {
        piece: i for i, piece in enumerate(Piece)
    },
    Colour.RED: {
        piece: i + 5 for i, piece in enumerate(Piece)
    },
}

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

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

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

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

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

    def get_rotation_hash(self, index, rotation):
        """
        Gets the random value for theon the given square.

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

        Returns:
            int: A 64-bit value.
        """
        rotation_index = rotation_lookup[rotation]
        return zobrist_table[index][rotation_index]
```

```
59
60     def apply_piece_hash(self, bitboard, piece, colour):
61         """
62         Updates the Zobrist hash with a new piece.
63
64         Args:
65             bitboard (int): The bitboard representation of the square.
66             piece (Piece): The piece on the square.
67             colour (Colour): The colour of the piece.
68         """
69         index = bitboard_to_index(bitboard)
70         piece_hash = self.get_piece_hash(index, piece, colour)
71         self.hash ^= piece_hash
72
73     def apply_rotation_hash(self, bitboard, rotation):
74         """Updates the Zobrist hash with a new rotation.
75
76         Args:
77             bitboard (int): The bitboard representation of the square.
78             rotation (Rotation): The rotation on the square.
79         """
80         index = bitboard_to_index(bitboard)
81         rotation_hash = self.get_rotation_hash(index, rotation)
82         self.hash ^= rotation_hash
83
84     def apply_red_move_hash(self):
85         """
86         Applies the Zobrist hash for the red player's move.
87         """
88         self.hash ^= red_move_hash
```

1.6.7 Cache

transposition_table.py

```
1  from data.constants import TranspositionFlag
2
3  class TranspositionEntry:
4      def __init__(self, score, move, flag, hash_key, depth):
5          self.score = score
6          self.move = move
7          self.flag = flag
8          self.hash_key = hash_key
9          self.depth = depth
10
11 class TranspositionTable:
12     def __init__(self, max_entries=100000):
13         self._max_entries = max_entries
14         self._table = dict()
15
16     def calculate_entry_index(self, hash_key):
17         """
18         Gets the dictionary key for a given Zobrist hash.
19
20         Args:
21             hash_key (int): A Zobrist hash.
22
23         Returns:
24             int: Key for the given hash.
25         """
26         # return hash_key % self._max_entries
```

```
27            return hash_key
28
29     def insert_entry(self, score, move, hash_key, depth, alpha, beta):
30         """
31         Inserts an entry into the transposition table.
32
33         Args:
34             score (int): The evaluation score.
35             move (Move): The best move found.
36             hash_key (int): The Zobrist hash key.
37             depth (int): The depth of the search.
38             alpha (int): The upper bound value.
39             beta (int): The lower bound value.
40
41         Raises:
42             Exception: Invalid depth or score.
43         """
44         if depth == 0 or alpha < score < beta:
45             flag = TranspositionFlag.EXACT
46             score = score
47         elif score <= alpha:
48             flag = TranspositionFlag.UPPER
49             score = alpha
50         elif score >= beta:
51             flag = TranspositionFlag.LOWER
52             score = beta
53         else:
54             raise Exception('(TranspositionTable.insert_entry)')
55
56         self._table[self.calculate_entry_index(hash_key)] = TranspositionEntry(
57     score, move, flag, hash_key, depth)
58
59         if len(self._table) > self._max_entries:
60             # Removes the longest-existing entry to free up space for more up-to-
60     date entries
61             # Expression to remove leftmost item taken from https://docs.python.
61     org/3/library/collections.html#ordereddict-objects
62             (k := next(iter(self._table)), self._table.pop(k))
62
63     def get_entry(self, hash_key, depth, alpha, beta):
64         """
65         Gets an entry from the transposition table.
66
67         Args:
68             hash_key (int): The Zobrist hash key.
69             depth (int): The depth of the search.
70             alpha (int): The alpha value for pruning.
71             beta (int): The beta value for pruning.
72
73         Returns:
74             tuple[int, Move] | tuple[None, None]: The evaluation score and the
74     best move found, if entry exists.
75         """
76         index = self.calculate_entry_index(hash_key)
77
78         if index not in self._table:
79             return None, None
80
81         entry = self._table[index]
82
83         if entry.hash_key == hash_key and entry.depth >= depth:
84             if entry.flag == TranspositionFlag.EXACT:
```

```
85                    return entry.score, entry.move
86
87            if entry.flag == TranspositionFlag.LOWER and entry.score >= beta:
88                    return entry.score, entry.move
89
90            if entry.flag == TranspositionFlag.UPPER and entry.score <= alpha:
91                    return entry.score, entry.move
92
93        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.
review.py

```
1  import pygame
2  from collections import deque
3  from data.states.game.components.capture_draw import CaptureDraw
4  from data.states.game.components.piece_group import PieceGroup
5  from data.constants import ReviewEventType, Colour, ShaderType
6  from data.states.game.components.laser_draw import LaserDraw
7  from data.utils.bitboard_helpers import bitboard_to_coords
8  from data.states.review.widget_dict import REVIEW_WIDGETS
9  from data.utils.browser_helpers import get_winner_string
10 from data.states.game.components.board import Board
11 from data.components.game_entry import GameEntry
12 from data.managers.logs import initialise_logger
13 from data.managers.window import window
14 from data.control import _State
15 from data.assets import MUSIC
16
17 logger = initialise_logger(__name__)
18
19 class Review(_State):
20     def __init__(self):
21         super().__init__()
22
23         self._moves = deque()
24         self._popped_moves = deque()
25         self._game_info = {}
26
27         self._board = None
28         self._piece_group = None
29         self._laser_draw = None
30         self._capture_draw = None
```

```python
31
32      def cleanup ( self ):
33          """
34          Cleanup  function .  Clears  shader  effects .
35          """
36          super () . cleanup ()
37
38          window . clear_apply_arguments ( ShaderType . BLOOM )
39          window . clear_effect ( ShaderType . RAYS )
40
41          return  None
42
43      def startup ( self ,  persist ):
44          """
45          Startup  function .  Initialises  all  objects ,  widgets  and  game  data .
46
47          Args:
48              persist  ( dict ):  Dict  containing  game  entry  data .
49          """
50          super () . startup ( REVIEW_WIDGETS ,  MUSIC [ 'review' ])
51
52          window . set_apply_arguments ( ShaderType . BASE ,  background_type = ShaderType .
    BACKGROUND_WAVES )
53          window . set_apply_arguments ( ShaderType . BLOOM ,  highlight_colours =[( pygame .
    Color ( '0x95e0cc' )). rgb ,  pygame . Color ( '0xf14e52' ). rgb ],  colour_intensity =0.8)
54          REVIEW_WIDGETS [ 'help' ]. kill ()
55
56          self . _moves  =  deque ( GameEntry . parse_moves ( persist . pop ( 'moves' ,  '' )))
57          self . _popped_moves  =  deque ()
58          self . _game_info  =  persist
59
60          self . _board  =  Board ( self . _game_info [ 'start_fen_string' ])
61          self . _piece_group  =  PieceGroup ()
62          self . _laser_draw  =  LaserDraw ( self . board_position ,  self . board_size )
63          self . _capture_draw  =  CaptureDraw ( self . board_position ,  self . board_size )
64
65          self . initialise_widgets ()
66          self . simulate_all_moves ()
67          self . refresh_pieces ()
68          self . refresh_widgets ()
69
70          self . draw ()
71
72      @property
73      def board_position ( self ):
74          return  REVIEW_WIDGETS [ 'chessboard' ]. position
75
76      @property
77      def board_size ( self ):
78          return  REVIEW_WIDGETS [ 'chessboard' ]. size
79
80      @property
81      def square_size ( self ):
82          return  self . board_size [0]  /  10
83
84      def initialise_widgets ( self ):
85          """
86          Initializes  the  widgets  for  a  new  game .
87          """
88          REVIEW_WIDGETS [ 'move_list' ]. reset_move_list ()
89          REVIEW_WIDGETS [ 'move_list' ]. kill ()
90          REVIEW_WIDGETS [ 'scroll_area' ]. set_image ()
```

81

```
 91
 92        REVIEW_WIDGETS['winner_text'].set_text(f'WINNER: {get_winner_string(self.
       _game_info["winner"])}')
 93        REVIEW_WIDGETS['blue_piece_display'].reset_piece_list()
 94        REVIEW_WIDGETS['red_piece_display'].reset_piece_list()
 95
 96        if self._game_info['time_enabled']:
 97            REVIEW_WIDGETS['timer_disabled_text'].kill()
 98        else:
 99            REVIEW_WIDGETS['blue_timer'].kill()
100            REVIEW_WIDGETS['red_timer'].kill()
101
102    def refresh_widgets(self):
103        """
104        Refreshes the widgets after every move.
105        """
106        REVIEW_WIDGETS['move_number_text'].set_text(f'MOVE NO: {(len(self._moves))
        / 2:.1f} / {(len(self._moves) + len(self._popped_moves)) / 2:.1f}')
107        REVIEW_WIDGETS['move_colour_text'].set_text(f'{self.calculate_colour().
       name} TO MOVE')
108
109        if self._game_info['time_enabled']:
110            if len(self._moves) == 0:
111                REVIEW_WIDGETS['blue_timer'].set_time(float(self._game_info['time'
       ]) * 60 * 1000)
112                REVIEW_WIDGETS['red_timer'].set_time(float(self._game_info['time'
       ]) * 60 * 1000)
113            else:
114                REVIEW_WIDGETS['blue_timer'].set_time(float(self._moves[-1]['
       blue_time']) * 60 * 1000)
115                REVIEW_WIDGETS['red_timer'].set_time(float(self._moves[-1]['
       red_time']) * 60 * 1000)
116
117        REVIEW_WIDGETS['scroll_area'].set_image()
118
119    def refresh_pieces(self):
120        """
121        Refreshes the pieces on the board.
122        """
123        self._piece_group.initialise_pieces(self._board.get_piece_list(), self.
       board_position, self.board_size)
124
125    def simulate_all_moves(self):
126        """
127        Simulates all moves at the start of every game to obtain laser results and
        fill up piece display and move list widgets.
128        """
129        for index, move_dict in enumerate(self._moves):
130            laser_result = self._board.apply_move(move_dict['move'], fire_laser=
       True)
131            self._moves[index]['laser_result'] = laser_result
132
133            if laser_result.hit_square_bitboard:
134                if laser_result.piece_colour == Colour.BLUE:
135                    REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.
       piece_hit)
136                elif laser_result.piece_colour == Colour.RED:
137                    REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.
       piece_hit)
138
139            REVIEW_WIDGETS['move_list'].append_to_move_list(move_dict['
       unparsed_move'])
```

```python
140
141     def calculate_colour(self):
142         """
143         Calculates the current active colour to move.
144
145         Returns:
146             Colour: The current colour to move.
147         """
148         if self._game_info['start_fen_string'][-1].lower() == 'b':
149             initial_colour = Colour.BLUE
150         elif self._game_info['start_fen_string'][-1].lower() == 'r':
151             initial_colour = Colour.RED
152
153         if len(self._moves) % 2 == 0:
154             return initial_colour
155         else:
156             return initial_colour.get_flipped_colour()
157
158     def handle_move(self, move, add_piece=True):
159         """
160         Handles applying or undoing a move.
161
162         Args:
163             move (dict): The move to handle.
164             add_piece (bool): Whether to add the captured piece to the display.
    Defaults to True.
165         """
166         laser_result = move['laser_result']
167         active_colour = self.calculate_colour()
168         self._laser_draw.add_laser(laser_result, laser_colour=active_colour)
169
170         if laser_result.hit_square_bitboard:
171             if laser_result.piece_colour == Colour.BLUE:
172                 if add_piece:
173                     REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result.
    piece_hit)
174                 else:
175                     REVIEW_WIDGETS['red_piece_display'].remove_piece(laser_result.
    piece_hit)
176             elif laser_result.piece_colour == Colour.RED:
177                 if add_piece:
178                     REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result.
    piece_hit)
179                 else:
180                     REVIEW_WIDGETS['blue_piece_display'].remove_piece(laser_result
    .piece_hit)
181
182             self._capture_draw.add_capture(
183                 laser_result.piece_hit,
184                 laser_result.piece_colour,
185                 laser_result.piece_rotation,
186                 bitboard_to_coords(laser_result.hit_square_bitboard),
187                 laser_result.laser_path[0][0],
188                 active_colour,
189                 shake=False
190             )
191
192     def update_laser_mask(self):
193         """
194         Updates the laser mask for the light rays effect.
195         """
196         temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
```

83

```
197            self._piece_group.draw(temp_surface)
198            mask = pygame.mask.from_surface(temp_surface, threshold=127)
199            mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255,
      0, 0, 255))
200
201            window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
202
203        def get_event(self, event):
204            """
205            Processes Pygame events.
206
207            Args:
208                event (pygame.event.Event): The event to handle.
209            """
210            if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
211                REVIEW_WIDGETS['help'].kill()
212
213            widget_event = self._widget_group.process_event(event)
214
215            if widget_event is None:
216                return
217
218            match widget_event.type:
219                case None:
220                    return
221
222                case ReviewEventType.MENU_CLICK:
223                    self.next = 'menu'
224                    self.done = True
225
226                case ReviewEventType.PREVIOUS_CLICK:
227                    if len(self._moves) == 0:
228                        return
229
230                    # Pop last applied move off first stack
231                    move = self._moves.pop()
232                    # Pushed onto second stack
233                    self._popped_moves.append(move)
234
235                    # Undo last applied move
236                    self._board.undo_move(move['move'], laser_result=move['
      laser_result'])
237                    self.handle_move(move, add_piece=False)
238                    REVIEW_WIDGETS['move_list'].pop_from_move_list()
239
240                    self.refresh_pieces()
241                    self.refresh_widgets()
242                    self.update_laser_mask()
243
244                case ReviewEventType.NEXT_CLICK:
245                    if len(self._popped_moves) == 0:
246                        return
247
248                    # Peek at second stack to get last undone move
249                    move = self._popped_moves[-1]
250
251                    # Reapply last undone move
252                    self._board.apply_move(move['move'])
253                    self.handle_move(move, add_piece=True)
254                    REVIEW_WIDGETS['move_list'].append_to_move_list(move['
      unparsed_move'])
255
```

```
256                 # Pop last undone move from second stack
257                 self._popped_moves.pop()
258                 # Push onto first stack
259                 self._moves.append(move)
260
261                 self.refresh_pieces()
262                 self.refresh_widgets()
263                 self.update_laser_mask()
264
265             case ReviewEventType.HELP_CLICK:
266                 self._widget_group.add(REVIEW_WIDGETS['help'])
267                 self._widget_group.handle_resize(window.size)
268
269     def handle_resize(self):
270         """
271         Handles resizing of the window.
272         """
273         super().handle_resize()
274         self._piece_group.handle_resize(self.board_position, self.board_size)
275         self._laser_draw.handle_resize(self.board_position, self.board_size)
276         self._capture_draw.handle_resize(self.board_position, self.board_size)
277
278         if self._laser_draw.firing:
279             self.update_laser_mask()
280
281     def draw(self):
282         """
283         Draws all components onto the window screen.
284         """
285         self._capture_draw.update()
286         self._widget_group.draw()
287         self._piece_group.draw(window.screen)
288         self._laser_draw.draw(window.screen)
289         self._capture_draw.draw(window.screen)
```

1.8 Database

This section outlines my database implementation using Python 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 their changes and datetime labelled 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 ,
20              time  REAL ,
21              number_of_ply  INTEGER  NOT  NULL ,
22              moves  TEXT  NOT  NULL
23          )
24      ''')
25
26      connection . commit ()
27      connection . close ()
28
29  def  downgrade ():
30      """
31      Downgrade  function  to  revert  table  creation .
32      """
33      connection = sqlite3 . connect ( database_path )
34      cursor = connection . cursor ()
35
36      cursor . execute ( '''
37          DROP  TABLE  games
38      ''')
39
40      connection . commit ()
41      connection . close ()
42
43  upgrade ()
44  # downgrade ()
```

Using the ALTER command allows me to rename table columns.

change_fen_string_column_name_23122024.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  rename  fen_string  column .
9       """
10      connection = sqlite3 . connect ( database_path )
11      cursor = connection . cursor ()
12
13      cursor . execute ( '''
14          ALTER  TABLE  games  RENAME  COLUMN  fen_string  TO  final_fen_string
15      ''')
16
17      connection . commit ()
18      connection . close ()
19
20  def  downgrade ():
21      """
22      Downgrade  function  to  revert  fen_string  column  renaming .
23      """
24      connection = sqlite3 . connect ( database_path )
25      cursor = connection . cursor ()
```

```
26
27      cursor.execute('''
28          ALTER TABLE games RENAME COLUMN final_fen_string TO fen_string
29      ''')
30
31      connection.commit()
32      connection.close()
33
34  upgrade()
35  # downgrade()
```

1.8.2 DML

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      """
```

```python
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 ):
```

```python
106             raise ValueError('(database_helpers.get_ordered_games) Invalid input
        arguments!')
107
108         connection = sqlite3.connect(database_path, detect_types=sqlite3.
        PARSE_DECLTYPES)
109         connection.row_factory = sqlite3.Row
110         cursor = connection.cursor()
111
112         # Match ascend bool to correct SQL keyword
113         if ascend:
114             ascend_arg = 'ASC'
115         else:
116             ascend_arg = 'DESC'
117
118         # Partition by winner, then order by time and number_of_ply
119         if column == 'winner':
120             cursor.execute(f'''
121                 SELECT * FROM
122                     (SELECT ROW_NUMBER() OVER (
123                         PARTITION BY winner
124                         ORDER BY time {ascend_arg}, number_of_ply {ascend_arg}
125                     ) AS row_num, * FROM games)
126                 WHERE row_num >= ? AND row_num <= ?
127             ''', (start_row, end_row))
128         else:
129         # Order by time or number_of_ply only
130             cursor.execute(f'''
131                 SELECT * FROM
132                     (SELECT ROW_NUMBER() OVER (
133                         ORDER BY {column} {ascend_arg}
134                     ) AS row_num, * FROM games)
135                 WHERE row_num >= ? AND row_num <= ?
136             ''', (start_row, end_row))
137
138         games = cursor.fetchall()
139
140         connection.close()
141
142         return [dict(game) for game in games]
143
144 def get_number_of_games():
145     """
146     Returns:
147         int: Number of rows in the games.
148     """
149     connection = sqlite3.connect(database_path)
150     cursor = connection.cursor()
151
152     cursor.execute("""
153         SELECT COUNT(ROWID) FROM games
154     """)
155
156     result = cursor.fetchall()[0][0]
157
158     connection.close()
159
160     return result
161
162 # delete_all_games()
```

1.9 Shaders

1.9.1 Shader Manager

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

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

shader.py

```python
1  from pathlib import Path
2  from array import array
3  import moderngl
4  from data.shaders.classes import shader_pass_lookup
5  from data.shaders.protocol import SMProtocol
6  from data.constants import ShaderType
7
8  shader_path = (Path(__file__).parent / '../shaders/').resolve()
9
10 SHADER_PRIORITY = [
11     ShaderType.CRT,
12     ShaderType.SHAKE,
13     ShaderType.BLOOM,
14     ShaderType.CHROMATIC_ABBREVIATION,
15     ShaderType.RAYS,
16     ShaderType.GRAYSCALE,
17     ShaderType.BASE,
18 ]
19
20 pygame_quad_array = array('f', [
21     -1.0, 1.0, 0.0, 0.0,
22     1.0, 1.0, 1.0, 0.0,
23     -1.0, -1.0, 0.0, 1.0,
24     1.0, -1.0, 1.0, 1.0,
25 ])
26
27 opengl_quad_array = array('f', [
28     -1.0, -1.0, 0.0, 0.0,
29     1.0, -1.0, 1.0, 0.0,
30     -1.0, 1.0, 0.0, 1.0,
31     1.0, 1.0, 1.0, 1.0,
32 ])
33
34 class ShaderManager(SMProtocol):
35     def __init__(self, ctx: moderngl.Context, screen_size):
36         self._ctx = ctx
37         self._ctx.gc_mode = 'auto'
38
39         self._screen_size = screen_size
40         self._opengl_buffer = self._ctx.buffer(data=opengl_quad_array)
41         self._pygame_buffer = self._ctx.buffer(data=pygame_quad_array)
42         self._shader_list = [ShaderType.BASE]
43
44         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
```

```python
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.
```

93

```
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`

```glsl
1  # version 330 core
2
3  in vec2 uvs;
4  out vec4 f_colour;
5
6  uniform sampler2D image;
7  uniform float threshold;
8  uniform float intensity;
9
10 void main() {
11     vec4 pixel = texture(image, uvs);
12     // Dot product used to calculate brightness of a pixel from its RGB values
13     // Values taken from https://en.wikipedia.org/wiki/Relative_luminance
14     float brightness = dot(pixel.rgb, vec3(0.2126, 0.7152, 0.0722));
15     float isBright = step(threshold, brightness);
16
17     f_colour = vec4(vec3(pixel.rgb * intensity) * isBright, 1.0);
18 }
```

Blur

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

`blur.py`

```python
1  from data.shaders.protocol import SMProtocol
2  from data.constants import ShaderType
3
4  BLUR_ITERATIONS = 4
5
6  class _Blur:
7      def __init__(self, shader_manager: SMProtocol):
8          self._shader_manager = shader_manager
9
10         shader_manager.create_framebuffer(ShaderType._BLUR)
11
12         shader_manager.create_framebuffer("blurPing")
13         shader_manager.create_framebuffer("blurPong")
14
15     def apply(self, texture):
16         """
17         Applies Gaussian blur to a given texture.
18
19         Args:
20             texture (moderngl.Texture): Texture to blur.
21         """
22         self._shader_manager.get_fbo_texture("blurPong").write(texture.read())
```

```
23
24          for _ in range ( BLUR_ITERATIONS ):
25              # Apply horizontal blur
26              self._shader_manager.render_to_fbo(
27                  ShaderType._BLUR ,
28                  texture = self._shader_manager.get_fbo_texture ( "blurPong" ),
29                  output_fbo = self._shader_manager.framebuffers [ "blurPing" ],
30                  passes =5 ,
31                  horizontal = True
32              )
33              # Apply vertical blur
34              self._shader_manager.render_to_fbo(
35                  ShaderType._BLUR ,
36                  texture = self._shader_manager.get_fbo_texture ( "blurPing" ), # Use
      horizontal blur result as input texture
37                  output_fbo = self._shader_manager.framebuffers [ "blurPong" ],
38                  passes =5 ,
39                  horizontal = False
40              )
41
42          self._shader_manager.render_to_fbo ( ShaderType._BLUR , self._shader_manager.
      get_fbo_texture ( "blurPong" ))
```

blur.frag

```
1  // Modified from https ://learnopengl.com/Advanced -Lighting/Bloom
2  #version 330 core
3
4  in vec2 uvs;
5  out vec4 f_colour;
6
7  uniform sampler2D image;
8  uniform bool horizontal;
9  uniform int passes;
10 uniform float weight [5] = float [] (0.227027 , 0.1945946 , 0.1216216 , 0.054054 ,
      0.016216);
11
12 void main () {
13     vec2 offset = 1.0 / textureSize ( image , 0);
14     vec3 result = texture ( image , uvs ).rgb * weight [0];
15
16     if ( horizontal ) {
17         for (int i = 1 ; i < passes ; ++i) {
18             result += texture ( image , uvs + vec2 ( offset.x * i, 0.0)).rgb * weight [i
      ];
19             result += texture ( image , uvs - vec2 ( offset.x * i, 0.0)).rgb * weight [i
      ];
20         }
21     }
22     else {
23         for (int i = 1 ; i < passes ; ++i) {
24             result += texture ( image , uvs + vec2 (0.0 , offset.y * i)).rgb * weight [i
      ];
25             result += texture ( image , uvs - vec2 (0.0 , offset.y * i)).rgb * weight [i
      ];
26         }
27     }
28
29     f_colour = vec4 ( result , 1.0);
30 }
```

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

```python
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.constants import ShaderType

BLOOM_INTENSITY = 0.6

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

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

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

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

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

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

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

        # Extract bright colours (highlights) from the texture
        _HighlightBrightness(self._shader_manager).apply(texture, intensity=
    brightness_intensity)
```

97

```
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

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.
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
3  #define PI 3.14159265
4
5  in vec2 uvs;
6  out vec4 f_colour;
7
8  uniform float softShadow;
9  uniform float resolution;
10 uniform float falloff;
11 uniform vec3 lightColour;
12 uniform vec2 angleClamp;
13 uniform sampler2D occlusionMap;
14 uniform sampler2D image;
15
16 vec3 normLightColour = lightColour / 255;
```

```glsl
17  vec2 radiansClamp = angleClamp * (PI / 180);
18
19  float sample(vec2 coord, float r) {
20    /*
21    Sample from the 1D distance map.
22
23    Returns:
24      float: 1.0 if sampled radius is greater than the passed radius, 0.0 if not.
25    */
26    return step(r, texture(image, coord).r);
27  }
28
29  void main() {
30    // Cartesian to polar transformation
31    // Range from [0, 1] -> [-1, 1]
32    vec2 norm = uvs.xy * 2.0 - 1.0;
33    float angle = atan(norm.y, norm.x);
34    float r = length(norm);
35
36    // The texture coordinates to sample our 1D lookup texture
37    // Always 0.0 on y-axis, as the texture is 1D
38    float x = (angle + PI) / (2.0 * PI); // Normalise angle to [0, 1]
39    vec2 tc = vec2(x, 0.0);
40
41    // Sample the 1D lookup texture to check if pixel is in light or in shadow
42    // Gives us hard shadows
43    // 1.0 -> in light, 0.0, -> in shadow
44    float inLight = sample(tc, r);
45    // Clamp angle so that only pixels within the range are in light
46    inLight = inLight * step(angle, radiansClamp.y) * step(radiansClamp.x, angle);
47
48    // Multiply the blur amount by the distance from the center
49    // So that the blurring increases as distance increases
50    float blur = (1.0 / resolution) * smoothstep(0.0, 0.1, r);
51
52    // Use gaussian blur to apply blur effecy
53    float sum = 0.0;
54
55    sum += sample(vec2(tc.x - blur * 4.0, tc.y), r) * 0.05;
56    sum += sample(vec2(tc.x - blur * 3.0, tc.y), r) * 0.09;
57    sum += sample(vec2(tc.x - blur * 2.0, tc.y), r) * 0.12;
58    sum += sample(vec2(tc.x - blur * 1.0, tc.y), r) * 0.15;
59
60    sum += inLight * 0.16;
61
62    sum += sample(vec2(tc.x + blur * 1.0, tc.y), r) * 0.15;
63    sum += sample(vec2(tc.x + blur * 2.0, tc.y), r) * 0.12;
64    sum += sample(vec2(tc.x + blur * 3.0, tc.y), r) * 0.09;
65    sum += sample(vec2(tc.x + blur * 4.0, tc.y), r) * 0.05;
66
67    // Mix with the softShadow uniform to toggle degree of softShadows
68    float finalLight = mix(inLight, sum, softShadow);
69
70    // Multiply the final light value with the distance, to give a radial falloff
71    // Use as the alpha value, with the light colour being the RGB values
72    f_colour = vec4(normLightColour, finalLight * smoothstep(1.0, falloff, r));
73  }
```

Class

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

`rays.py`

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

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

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

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

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

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

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

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

            # Apply lightmap shader, shadowmap and occlusion are included within
```

```
         the _Lightmap class
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 )
```