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.

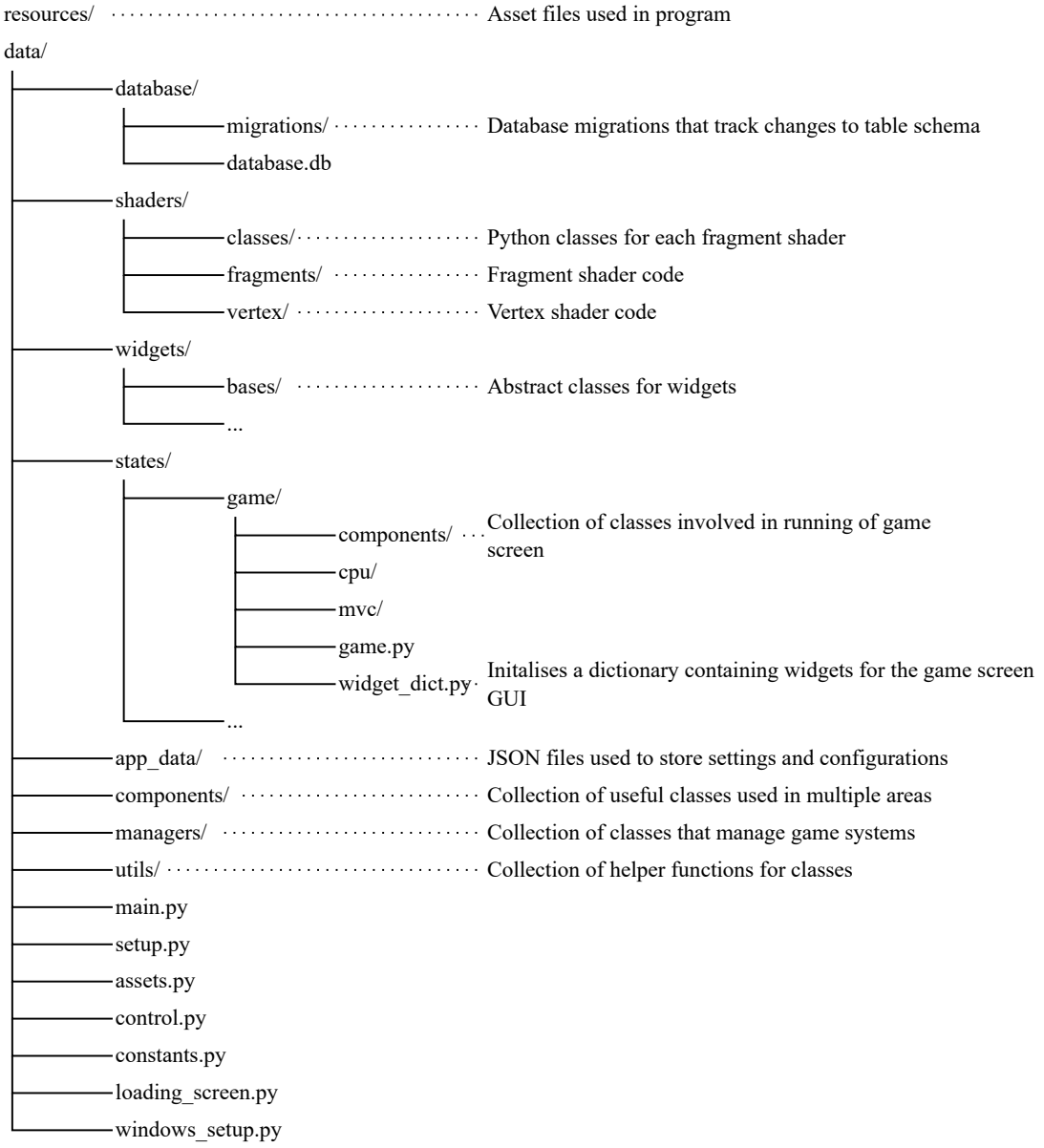

Figure 1.1: File tree diagram

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 1.9.4)
- 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`

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

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

from data.loading_screen import LoadingScreen

states = [None, None]

def load_states():
    """
    Initialises instances of all screens, executed on another thread with results
    being stored to the main thread by modifying a mutable such as the states list
    """
    from data.control import Control
    from data.states.game.game import Game
    from data.states.menu.menu import Menu
```

```
    from data.states.settings.settings import Settings
    from data.states.config.config import Config
    from data.states.browser.browser import Browser
    from data.states.review.review import Review
    from data.states.editor.editor import Editor

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

    app = Control()

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

loading_screen = LoadingScreen(load_states)

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

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

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

1.3.2 Loading Screen

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

loading_screen.py

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

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

def easeOutBack(progress):
```

```
    """
    Represents a cubic function for easing the logo position.
    Starts quickly and has small overshoot, then ends slowly.

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

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

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

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

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

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

        self.run()

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

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

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

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

    def event_loop(self):
        """
        Handles events for the loading screen, no user input is taken except to
    quit the game.
        """
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
```

```
                pygame.quit()
                sys.exit()

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

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

        window.update()

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

1.3.3 Helper functions

These files provide useful functions for different classes.

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

```
import pygame
from PIL import Image
from functools import cache
from random import sample, randint
import math

@cache
def scale_and_cache(image, target_size):
    """
    Caches image when resized repeatedly.

    Args:
        image (pygame.Surface): Image surface to be resized.
        target_size (tuple[float, float]): New image size.

    Returns:
        pygame.Surface: Resized image surface.
    """
    return pygame.transform.scale(image, target_size)

@cache
def smoothscale_and_cache(image, target_size):
    """
    Same as scale_and_cache, but with the Pygame smoothscale function.

    Args:
        image (pygame.Surface): Image surface to be resized.
        target_size (tuple[float, float]): New image size.

    Returns:
        pygame.Surface: Resized image surface.
    """
```

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

```
    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):
```

```
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
```

```
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():
```

```
    return load_json(user_file_path)

def get_default_settings():
    return load_json(default_file_path)

def get_themes():
    return load_json(themes_file_path)

def update_user_settings(data):
    """
    Rewrites JSON file for user settings with new data.

    Args:
        data (dict): Dictionary storing updated user settings.

    Raises:
        Exception: Invalid file.
    """
    try:
        with open(user_file_path, 'w') as f:
            json.dump(data, f, indent=4)
    except:
        raise Exception('Invalid JSON file (data_helpers.py)')
```

widget_helpers.py (Files used for creating widgets)

```
import pygame
from math import sqrt

def create_slider(size, fill_colour, border_width, border_colour):
    """
    Creates surface for sliders.

    Args:
        size (list[int, int]): Image size.
        fill_colour (pygame.Color): Fill (inner) colour.
        border_width (float): Border width.
        border_colour (pygame.Color): Border colour.

    Returns:
        pygame.Surface: Slider image surface.
    """
    gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
    border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
    height))

    # Draws rectangle with a border radius half of image height, to draw an
    rectangle with semicurclar cap (obround)
    pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int
    (size[1] / 2))
    pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
    border_width), border_radius=int(size[1] / 2))

    return gradient_surface

def create_slider_gradient(size, border_width, border_colour):
    """
    Draws surface for colour slider, with a full colour gradient as fill colour.

    Args:
        size (list[int, int]): Image size.
```

```
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
```

```
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
```

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

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

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

        Args:
            arg (str): Attribute name.

        Raises:
            KeyError: Instance does not have requested attribute.

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

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

        return item

theme = ThemeManager()
```

1.4 GUI

1.4.1 Laser

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

laser_draw.py

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

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

GLOW_SCALE_FACTOR = 1.5

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

```
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                      0.5,
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
```

```
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]
```

```
            vertex_2 = points_list[i + 1]
            vector_1 = get_vector(center, vertex_1)
            vector_2 = get_vector(center, vertex_2)
            angle = get_angle_between_vectors(vector_1, vector_2)

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

            corners_to_draw = None

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

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

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

            else:
                corners_to_draw = 1

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

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

            fragment_list.append(cropped_image)

        return fragment_list

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

        Args:
            piece (Piece): The piece type.
            colour (Colour): The active colour of the piece.
            rotation (int): The rotation of the piece.
            position (tuple[int, int]): The position where particles originate from.
            size (tuple[int, int]): The size of the piece.
        """
        piece_sprite = PieceSprite(piece, colour, rotation)
        piece_sprite.set_geometry((0, 0), size)
        piece_sprite.set_image()

        particles = self.fragment_image(piece_sprite.image, 5)

        for particle in particles:
            self.add_particle(particle, position)

    def add_sparks(self, radius, colour, position):
```

```
96          """
97          Adds laser spark particles.
98
99          Args:
100             radius (int): The radius of the sparks.
101             colour (Colour): The active colour of the sparks.
102             position (tuple[int, int]): The position where particles originate
    from.
103         """
104         for i in range(randint(10, 15)):
105             velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
106             random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in
    colour]
107             self._particles.append([None, [radius, random_colour], [*position],
    velocity, 0])
108
109     def add_particle(self, image, position):
110         """
111         Adds a particle.
112
113         Args:
114             image (pygame.Surface): The image of the particle.
115             position (tuple): The position of the particle.
116         """
117         velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
118
119         # Each particle is stored with its attributes: [surface, copy of surface,
    position, velocity, lifespan]
120         self._particles.append([image, image.copy(), [*position], velocity, 0])
121
122     def update(self):
123         """
124         Updates each particle and its attributes.
125         """
126         for i in range(len(self._particles) - 1, -1, -1):
127             particle = self._particles[i]
128
129             #update position
130             particle[2][0] += particle[3][0]
131             particle[2][1] += particle[3][1]
132
133             #update lifespan
134             self._particles[i][4] += 0.01
135
136             if self._particles[i][4] >= 1:
137                 self._particles.pop(i)
138                 continue
139
140             if isinstance(particle[1], pygame.Surface): # Particle is a piece
141                 # Update velocity
142                 particle[3][1] += self._gravity
143
144                 # Update size
145                 image_size = particle[1].get_rect().size
146                 end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink)
     * image_size[1])
147                 target_size = (image_size[0] - particle[4] * (image_size[0] -
    end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))
148
149                 # Update rotation
150                 rotation = (self._rotation if particle[3][0] <= 0 else -self.
    _rotation) * particle[4]
```

```
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
```

```
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'
```

```
60          self._anchor_y = kwargs.get('anchor_y') or 'top'
61          self._fixed_position = kwargs.get('fixed_position')
62          scale_mode = kwargs.get('scale_mode') or 'both'
63
64          if kwargs.get('relative_size'):
65              match scale_mode:
66                  case 'height':
67                      self._relative_size = kwargs.get('relative_size')
68                  case 'width':
69                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] *
    self.surface_size[0]) / self.surface_size[1])
70                  case 'both':
71                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
72                  case _:
73                      raise ValueError('(_Widget.__init__) Unknown scale mode:',
    scale_mode)
74          else:
75              self._relative_size = (1, 1)
76
77          if 'margin' in kwargs:
78              self._relative_margin = kwargs.get('margin') / self._raw_surface_size
    [1]
79
80              if (self._relative_margin * 2) > min(self._relative_size[0], self.
    _relative_size[1]):
81                  raise ValueError('(_Widget.__init__) Margin larger than specified
    size!')
82
83          if 'border_width' in kwargs:
84              self._relative_border_width = kwargs.get('border_width') / self.
    _raw_surface_size[1]
85
86          if 'border_radius' in kwargs:
87              self._relative_border_radius = kwargs.get('border_radius') / self.
    _raw_surface_size[1]
88
89          if 'border_colour' in kwargs:
90              self._border_colour = pygame.Color(kwargs.get('border_colour'))
91
92          if 'fill_colour' in kwargs:
93              self._fill_colour = pygame.Color(kwargs.get('fill_colour'))
94
95          if 'text_colour' in kwargs:
96              self._text_colour = pygame.Color(kwargs.get('text_colour'))
97
98          if 'font' in kwargs:
99              self._font = kwargs.get('font')
100
101     @property
102     def surface_size(self):
103         """
104         Gets the size of the surface widget is drawn on.
105         Can be either the window size, or another widget size if assigned to a
    parent.
106
107         Returns:
108             tuple[int, int]: The size of the surface.
109         """
110         if self._parent:
111             return self._parent.size
```

```
112             else:
113                 return self._raw_surface_size
114 
115         @property
116         def position(self):
117             """
118             Gets the position of the widget.
119             Accounts for fixed position attribute, where widget is positioned in
    pixels regardless of screen size.
120             Acounts for anchor direction, where position attribute is calculated
    relative to one side of the screen.
121 
122             Returns:
123                 tuple[int, int]: The position of the widget.
124             """
125             x, y = None, None
126             if self._fixed_position:
127                 x, y = self._fixed_position
128             if x is None:
129                 x = self._relative_position[0] * self.surface_size[0]
130             if y is None:
131                 y = self._relative_position[1] * self.surface_size[1]
132 
133             if self._anchor_x == 'left':
134                 x = x
135             elif self._anchor_x == 'right':
136                 x = self.surface_size[0] - x - self.size[0]
137             elif self._anchor_x == 'center':
138                 x = (self.surface_size[0] / 2 - self.size[0] / 2) + x
139 
140             if self._anchor_y == 'top':
141                 y = y
142             elif self._anchor_y == 'bottom':
143                 y = self.surface_size[1] - y - self.size[1]
144             elif self._anchor_y == 'center':
145                 y = (self.surface_size[1] / 2 - self.size[1] / 2) + y
146 
147             # Position widget relative to parent, if exists.
148             if self._parent:
149                 return (x + self._parent.position[0], y + self._parent.position[1])
150             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
```

```
class _Circular:
    def __init__(self, items_dict, **kwargs):
        # The key, value pairs are stored within a dictionary, while the keys to access them are stored within circular linked list.
        self._items_dict = items_dict
        self._keys_list = CircularLinkedList(list(items_dict.keys()))

    @property
    def current_key(self):
        """
        Gets the current head node of the linked list, and returns a key stored as the node data.
        Returns:
            Data of linked list head.
        """
        return self._keys_list.get_head().data

    @property
    def current_item(self):
        """
        Gets the value in self._items_dict with the key being self.current_key.

        Returns:
            Value stored with key being current head of linked list.
        """
        return self._items_dict[self.current_key]

    def set_next_item(self):
        """
        Sets the next item in as the current item.
        """
        self._keys_list.shift_head()

    def set_previous_item(self):
        """
        Sets the previous item as the current item.
        """
        self._keys_list.unshift_head()

    def set_to_key(self, key):
        """
        Sets the current item to the specified key.

        Args:
            key: The key to set as the current item.

        Raises:
            ValueError: If no nodes within the circular linked list contains the key as its data.
        """
        if self._keys_list.data_in_list(key) is False:
            raise ValueError('(_Circular.set_to_key) Key not found:', key)

        for _ in range(len(self._items_dict)):
            if self.current_key == key:
                self.set_image()
                self.set_geometry()
                return

            self.set_next_item()
```

Circular Linked List

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

circular_linked_list.py

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

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

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

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

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

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

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

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

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

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

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

```
            new_node.previous = self._head.previous
            self._head.previous.next = new_node
            self._head.previous = new_node

            self._head = new_node

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

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

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

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

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

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

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

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

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

    def delete(self, data):
        """
```

```
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:

- 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`

```
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'],
13     SettingsEventType.DROPDOWN_CLICK: ['selected_word'],
```

```
    SettingsEventType.VOLUME_SLIDER_CLICK: ['volume', 'volume_type'],
    SettingsEventType.SHADER_PICKER_CLICK: ['data'],
    SettingsEventType.PARTICLES_CLICK: ['toggled'],
    SettingsEventType.OPENGL_CLICK: ['toggled'],
    ConfigEventType.TIME_TYPE: ['time'],
    ConfigEventType.FEN_STRING_TYPE: ['time'],
    ConfigEventType.CPU_DEPTH_CLICK: ['data'],
    ConfigEventType.PVC_CLICK: ['data'],
    ConfigEventType.PRESET_CLICK: ['fen_string'],
    BrowserEventType.BROWSER_STRIP_CLICK: ['selected_index'],
    BrowserEventType.PAGE_CLICK: ['data'],
    EditorEventType.PICK_PIECE_CLICK: ['piece', 'active_colour'],
    EditorEventType.ROTATE_PIECE_CLICK: ['rotation_direction'],
}

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

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

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

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

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

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

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

            return event_cls(event_type, **kwargs)

        else:
            return event_cls(event_type)
```

ReactiveIconButton

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

reactive_icon_button.py

```
from data.widgets.reactive_button import ReactiveButton
from data.constants import WidgetState
from data.widgets.icon import Icon

class ReactiveIconButton(ReactiveButton):
    def __init__(self, base_icon, hover_icon, press_icon, **kwargs):
        # Composition is used here, to initialise the Icon widgets for each widget state
        widgets_dict = {
            WidgetState.BASE: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=base_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.HOVER: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=hover_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.PRESS: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=press_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            )
        }

        super().__init__(
            widgets_dict=widgets_dict,
            **kwargs
        )
```

ReactiveButton

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

reactive_button.py

```
from data.components.custom_event import CustomEvent
from data.widgets.bases.pressable import _Pressable
from data.widgets.bases.circular import _Circular
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState

class ReactiveButton(_Pressable, _Circular, _Widget):
```

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

        self._center = center

        self.initialise_new_colours(self._fill_colour)

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

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

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

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

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

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

        Args:
            new_surface_size (list[int, int]): New surface size.
        """
        super().set_surface_size(new_surface_size)
```

```
65          for item in self._items_dict.values():
66              item.set_surface_size(new_surface_size)
67
68      def process_event(self, event):
69          """
70          Processes Pygame events.
71
72          Args:
73              event (pygame.Event): Event to process.
74
75          Returns:
76              CustomEvent: CustomEvent of current item, with current key included
77          """
78          widget_event = super().process_event(event)
79          self.current_item.process_event(event)
80
81          if widget_event:
82              return CustomEvent(**vars(widget_event), data=self.current_key)
```

ColourSlider

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

`colour_slider.py`

```
1  import pygame
2  from data.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
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()
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]
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                         logger.info('COPIED')
222
223                     elif event.key == pygame.K_v:
224                         pasted_text = pyperclip.paste()
225                         pasted_text = ''.join(char for char in pasted_text if 32
    <= ord(char) <= 127)
226                         self._text = self._text[:self._cursor_index] + pasted_text
     + self._text[self._cursor_index:]
227                         self._cursor_index += len(pasted_text)
228
229                     self.resize_text()
230                     self.set_image()
231                     self.set_geometry()
232
233                     return
234
235                 match event.key:
```

```
                    case pygame.K_BACKSPACE:
                        if self._cursor_index > 0:
                            self._text = self._text[:self._cursor_index - 1] + self._text[self._cursor_index:]
                        self._cursor_index = max(0, self._cursor_index - 1)

                    case pygame.K_RIGHT:
                        self._cursor_index = min(len(self._text), self._cursor_index + 1)

                    case pygame.K_LEFT:
                        self._cursor_index = max(0, self._cursor_index - 1)

                    case pygame.K_ESCAPE:
                        self.unfocus_input()
                        return CustomEvent(**vars(self._event), text=self.text)

                    case pygame.K_RETURN:
                        self.unfocus_input()
                        return CustomEvent(**vars(self._event), text=self.text)

                    case _:
                        if not event.unicode:
                            return

                        potential_text = self._text[:self._cursor_index] + event.unicode + self._text[self._cursor_index:]

                        # Validator lambda function used to check if inputted text is valid before displaying
                        # e.g. Time control input has a validator function checking if text represents a float
                        if self._validator(potential_text) is False:
                            return

                        self._text = potential_text
                        self._cursor_index += 1

                self._blinking_cooldown += 1
                animation.set_timer(500, lambda: self.subtract_blinking_cooldown(1))

                self.resize_text()
                self.set_image()
                self.set_geometry()

    def subtract_blinking_cooldown(self, cooldown):
        """
        Subtracts blinking cooldown after certain timeframe. When blinking_cooldown is 1, cursor is able to be drawn.

        Args:
            cooldown (float): Duration before cursor can no longer be drawn.
        """
        self._blinking_cooldown = self._blinking_cooldown - cooldown

    def set_image(self):
        """
        Draws text input widget to image.
        """
        super().set_image()

```

```
291         if self._cursor_index is not None:
292             scaled_cursor = pygame.transform.scale(self._empty_cursor, self.
    cursor_size)
293             scaled_cursor.fill(self._cursor_colour)
294             self.image.blit(scaled_cursor, self.cursor_position)
295 
296     def update(self):
297         """
298         Overrides based update method, to handle cursor blinking.
299         """
300         super().update()
301         # Calculate if cursor should be shown or not
302         cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
303         if cursor_frame == 1 and self._blinking_cooldown == 0:
304             self._cursor_colour = (0, 0, 0, 0)
305         else:
306             self._cursor_colour = self._cursor_colour_copy
307         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.states.game.cpu.engines import ABMinimaxCPU
6  from data.components.custom_event import CustomEvent
7  from data.utils.bitboard_helpers import is_occupied
8  from data.states.game.components.board import Board
9  from data.utils import input_helpers as ip_helpers
10 from data.states.game.components.move import Move
11 from data.managers.logs import initialise_logger
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 
24         self.states = {
25             'CPU_ENABLED': game_config['CPU_ENABLED'],
26             'CPU_DEPTH': game_config['CPU_DEPTH'],
27             'AWAITING_CPU': False,
28             'WINNER': None,
29             'PAUSED': False,
30             'ACTIVE_COLOUR': game_config['COLOUR'],
31             'TIME_ENABLED': game_config['TIME_ENABLED'],
32             'TIME': game_config['TIME'],
33             'START_FEN_STRING': game_config['FEN_STRING'],
34             'MOVES': [],
35             'ZOBRIST_KEYS': []
```

```
36          }
37
38          self._cpu = ABMinimaxCPU(self.states['CPU_DEPTH'], self.cpu_callback,
    verbose=False)
39          self._cpu_thread = CPUThread(self._cpu)
40          self._cpu_thread.start()
41          self._cpu_move = None
42
43          logger.info(f'Initialising CPU depth of {self.states['CPU_DEPTH']}')
44
45      def register_listener(self, listener, parent_class):
46          """
47          Registers listener method of another MVC class.
48
49          Args:
50              listener (callable): Listener callback function.
51              parent_class (str): Class name.
52          """
53          self._listeners[parent_class].append(listener)
54
55      def alert_listeners(self, event):
56          """
57          Alerts all registered classes of an event by calling their listener
    function.
58
59          Args:
60              event (GameEventType): Event to pass as argument.
61
62          Raises:
63              Exception: If an unrecgonised event tries to be passed onto listeners.
64          """
65          for parent_class, listeners in self._listeners.items():
66              match event.type:
67                  case GameEventType.UPDATE_PIECES:
68                      if parent_class in 'game':
69                          for listener in listeners: listener(event)
70
71                  case GameEventType.SET_LASER:
72                      if parent_class == 'game':
73                          for listener in listeners: listener(event)
74
75                  case GameEventType.PAUSE_CLICK:
76                      if parent_class in ['pause', 'game']:
77                          for listener in listeners:
78                              listener(event)
79
80                  case _:
81                      raise Exception('Unhandled event type (GameModel.
    alert_listeners)')
82
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))
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         """
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.components.cursor import Cursor
14 from data.managers.window import window
15 from data.managers.audio import audio
16 from data.assets import SFX
17
18 class GameView:
19     def __init__(self, model):
20         self._model = model
21         self._hide_pieces = False
22         self._selected_coords = None
23         self._event_to_func_map = {
24             GameEventType.UPDATE_PIECES: self.handle_update_pieces,
25             GameEventType.SET_LASER: self.handle_set_laser,
26             GameEventType.PAUSE_CLICK: self.handle_pause,
27         }
28
29         # Register model event handling with process_model_event()
```

```
30          self._model.register_listener(self.process_model_event, 'game')
31
32          # Initialise WidgetGroup with map of widgets
33          self._widget_group = WidgetGroup(GAME_WIDGETS)
34          self._widget_group.handle_resize(window.size)
35          self.initialise_widgets()
36
37          self._cursor = Cursor()
38          self._laser_draw = LaserDraw(self.board_position, self.board_size)
39          self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
40          self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
41          self._capture_draw = CaptureDraw(self.board_position, self.board_size)
42          self._piece_group = PieceGroup()
43          self.handle_update_pieces()
44
45          self.set_status_text(StatusText.PLAYER_MOVE)
46
47      @property
48      def board_position(self):
49          return GAME_WIDGETS['chessboard'].position
50
51      @property
52      def board_size(self):
53          return GAME_WIDGETS['chessboard'].size
54
55      @property
56      def square_size(self):
57          return self.board_size[0] / 10
58
59      def initialise_widgets(self):
60          """
61          Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
62          """
63          GAME_WIDGETS['move_list'].reset_move_list()
64          GAME_WIDGETS['move_list'].kill()
65          GAME_WIDGETS['help'].kill()
66          GAME_WIDGETS['tutorial'].kill()
67
68          GAME_WIDGETS['scroll_area'].set_image()
69
70          GAME_WIDGETS['chessboard'].refresh_board()
71
72          GAME_WIDGETS['blue_piece_display'].reset_piece_list()
73          GAME_WIDGETS['red_piece_display'].reset_piece_list()
74
75      def set_status_text(self, status):
76          """
77          Sets text on status text widget.
78
79          Args:
80              status (StatusText): The game stage for which text should be displayed
     for.
81          """
82          match status:
83              case StatusText.PLAYER_MOVE:
84                  GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
    ACTIVE_COLOUR'].name}'s turn to move")
85              case StatusText.CPU_MOVE:
86                  GAME_WIDGETS['status_text'].set_text(f"CPU calculating a crazy
    move...")
87              case StatusText.WIN:
88                  if self._model.states['WINNER'] == Miscellaneous.DRAW:
```

```
89                      GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring
    ...")
90                  else:
91                      GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
    WINNER'].name} won!")
92              case StatusText.DRAW:
93                  GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring...")
94
95      def handle_resize(self):
96          """
97          Handles resizing of the window.
98          """
99          self._overlay_draw.handle_resize(self.board_position, self.board_size)
100         self._capture_draw.handle_resize(self.board_position, self.board_size)
101         self._piece_group.handle_resize(self.board_position, self.board_size)
102         self._laser_draw.handle_resize(self.board_position, self.board_size)
103         self._laser_draw.handle_resize(self.board_position, self.board_size)
104         self._widget_group.handle_resize(window.size)
105
106         if self._laser_draw.firing:
107             self.update_laser_mask()
108
109     def handle_update_pieces(self, event=None):
110         """
111         Callback function to update pieces after move.
112
113         Args:
114             event (GameEventType, optional): If updating pieces after player move,
     event contains move information. Defaults to None.
115             toggle_timers (bool, optional): Toggle timers on and off for new
    active colour. Defaults to True.
116         """
117         piece_list = self._model.get_piece_list()
118         self._piece_group.initialise_pieces(piece_list, self.board_position, self.
    board_size)
119
120         if event:
121             GAME_WIDGETS['move_list'].append_to_move_list(event.move_notation)
122             GAME_WIDGETS['scroll_area'].set_image()
123             audio.play_sfx(SFX['piece_move'])
124
125         if self._model.states['ACTIVE_COLOUR'] == Colour.BLUE:
126             self.set_status_text(StatusText.PLAYER_MOVE)
127         elif self._model.states['CPU_ENABLED'] is False:
128             self.set_status_text(StatusText.PLAYER_MOVE)
129         else:
130             self.set_status_text(StatusText.CPU_MOVE)
131
132         if self._model.states['WINNER'] is not None:
133             self.toggle_timer(self._model.states['ACTIVE_COLOUR'], False)
134             self.toggle_timer(self._model.states['ACTIVE_COLOUR'].
    get_flipped_colour(), False)
135
136             self.set_status_text(StatusText.WIN)
137
138             audio.play_sfx(SFX['sphinx_destroy_1'])
139             audio.play_sfx(SFX['sphinx_destroy_2'])
140             audio.play_sfx(SFX['sphinx_destroy_3'])
141
142     def handle_set_laser(self, event):
143         """
144         Callback function to draw laser after move.
```

```
145
146         Args:
147             event (GameEventType): Contains laser trajectory information.
148         """
149         laser_result = event.laser_result
150
151         # If laser has hit a piece
152         if laser_result.hit_square_bitboard:
153             coords_to_remove = bitboard_to_coords(laser_result.hit_square_bitboard
    )
154             self._piece_group.remove_piece(coords_to_remove)
155
156             if laser_result.piece_colour == Colour.BLUE:
157                 GAME_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit
    )
158             elif laser_result.piece_colour == Colour.RED:
159                 GAME_WIDGETS['blue_piece_display'].add_piece(laser_result.
    piece_hit)
160
161             # Draw piece capture GFX
162             self._capture_draw.add_capture(
163                 laser_result.piece_hit,
164                 laser_result.piece_colour,
165                 laser_result.piece_rotation,
166                 coords_to_remove,
167                 laser_result.laser_path[0][0],
168                 self._model.states['ACTIVE_COLOUR']
169             )
170
171         self._laser_draw.add_laser(laser_result, self._model.states['ACTIVE_COLOUR
    '])
172         self.update_laser_mask()
173
174     def handle_pause(self, event=None):
175         """
176         Callback function for pausing timer.
177
178         Args:
179             event (None): Event argument not used.
180         """
181         is_active = not(self._model.states['PAUSED'])
182         self.toggle_timer(self._model.states['ACTIVE_COLOUR'], is_active)
183
184     def initialise_timers(self):
185         """
186         Initialises both timers with the correct amount of time and starts the
    timer for the active colour.
187         """
188         if self._model.states['TIME_ENABLED']:
189             GAME_WIDGETS['blue_timer'].set_time(self._model.states['TIME'] * 60 *
    1000)
190             GAME_WIDGETS['red_timer'].set_time(self._model.states['TIME'] * 60 *
    1000)
191         else:
192             GAME_WIDGETS['blue_timer'].kill()
193             GAME_WIDGETS['red_timer'].kill()
194
195         self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
196
197     def toggle_timer(self, colour, is_active):
198         """
199         Stops or resumes timer.
```

```
200
201             Args:
202                 colour (Colour): Timer to toggle.
203                 is_active (bool): Whether to pause or resume timer.
204             """
205             if colour == Colour.BLUE:
206                 GAME_WIDGETS['blue_timer'].set_active(is_active)
207             elif colour == Colour.RED:
208                 GAME_WIDGETS['red_timer'].set_active(is_active)
209
210         def update_laser_mask(self):
211             """
212             Uses pygame.mask to create a mask for the pieces.
213             Used for occluding the ray shader.
214             """
215             temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
216             self._piece_group.draw(temp_surface)
217             mask = pygame.mask.from_surface(temp_surface, threshold=127)
218             mask_surface = mask.to_surface(unsetcolor=(0, 0, 0, 255), setcolor=(255,
        0, 0, 255))
219
220             window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
221
222         def draw(self):
223             """
224             Draws GUI and pieces onto the screen.
225             """
226             self._widget_group.update()
227             self._capture_draw.update()
228
229             self._widget_group.draw()
230             self._overlay_draw.draw(window.screen)
231
232             if self._hide_pieces is False:
233                 self._piece_group.draw(window.screen)
234
235             self._laser_draw.draw(window.screen)
236             self._drag_and_drop.draw(window.screen)
237             self._capture_draw.draw(window.screen)
238
239         def process_model_event(self, event):
240             """
241             Registered listener function for handling GameModel events.
242             Each event is mapped to a callback function, and the appropiate one is run
        .
243
244             Args:
245                 event (GameEventType): Game event to process.
246
247             Raises:
248                 KeyError: If an unrecgonised event type is passed as the argument.
249             """
250             try:
251                 self._event_to_func_map.get(event.type)(event)
252             except:
253                 raise KeyError('Event type not recognized in Game View (GameView.
        process_model_event):', event.type)
254
255         def set_overlay_coords(self, available_coords_list, selected_coord):
256             """
257             Set board coordinates for potential moves overlay.
258
```

```
259         Args:
260             available_coords_list (list[tuple[int, int]], ...): Array of
    coordinates
261             selected_coord (list[int, int]): Coordinates of selected piece.
262         """
263         self._selected_coords = selected_coord
264         self._overlay_draw.set_selected_coords(selected_coord)
265         self._overlay_draw.set_available_coords(available_coords_list)
266 
267     def get_selected_coords(self):
268         return self._selected_coords
269 
270     def set_dragged_piece(self, piece, colour, rotation):
271         """
272         Passes information of the dragged piece to the dragging drawing class.
273 
274         Args:
275             piece (Piece): Piece type of dragged piece.
276             colour (Colour): Colour of dragged piece.
277             rotation (Rotation): Rotation of dragged piece.
278         """
279         self._drag_and_drop.set_dragged_piece(piece, colour, rotation)
280 
281     def remove_dragged_piece(self):
282         """
283         Stops drawing dragged piece when user lets go of piece.
284         """
285         self._drag_and_drop.remove_dragged_piece()
286 
287     def convert_mouse_pos(self, event):
288         """
289         Passes information of what mouse cursor is interacting with to a
    GameController object.
290 
291         Args:
292             event (pygame.Event): Mouse event to process.
293 
294         Returns:
295             CustomEvent | None: Contains information what mouse is doing.
296         """
297         clicked_coords = screen_pos_to_coords(event.pos, self.board_position, self
    .board_size)
298 
299         if event.type == pygame.MOUSEBUTTONDOWN:
300             if clicked_coords:
301                 return CustomEvent.create_event(GameEventType.BOARD_CLICK, coords=
    clicked_coords)
302 
303             else:
304                 return None
305 
306         elif event.type == pygame.MOUSEBUTTONUP:
307             if self._drag_and_drop.dragged_sprite:
308                 piece, colour, rotation = self._drag_and_drop.get_dragged_info()
309                 piece_dragged = self._drag_and_drop.remove_dragged_piece()
310                 return CustomEvent.create_event(GameEventType.PIECE_DROP, coords=
    clicked_coords, piece=piece, colour=colour, rotation=rotation, remove_overlay=
    piece_dragged)
311 
312     def add_help_screen(self):
313         """
314         Draw help overlay when player clicks on the help button.
```

```
315         """
316         self._widget_group.add(GAME_WIDGETS['help'])
317         self._widget_group.handle_resize(window.size)
318 
319     def add_tutorial_screen(self):
320         """
321         Draw tutorial overlay when player clicks on the tutorial button.
322         """
323         self._widget_group.add(GAME_WIDGETS['tutorial'])
324         self._widget_group.handle_resize(window.size)
325         self._hide_pieces = True
326 
327     def remove_help_screen(self):
328         GAME_WIDGETS['help'].kill()
329 
330     def remove_tutorial_screen(self):
331         GAME_WIDGETS['tutorial'].kill()
332         self._hide_pieces = False
333 
334     def process_widget_event(self, event):
335         """
336         Passes Pygame event to WidgetGroup to allow individual widgets to process
    events.
337 
338         Args:
339             event (pygame.Event): Event to process.
340 
341         Returns:
342             CustomEvent | None: A widget event.
343         """
344         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
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_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_new_game = to_new_game
18 
19         self._view.initialise_timers()
20 
21     def cleanup(self, next):
22         """
23         Handles game quit, either leaving to main menu or restarting a new game.
24 
25         Args:
```

```
26              next (str): New state to switch to.
27          """
28          self._model.kill_thread()
29
30          if next == 'menu':
31              self._to_menu()
32          elif next == 'game':
33              self._to_new_game()
34
35      def make_move(self, move):
36          """
37          Handles player move.
38
39          Args:
40              move (Move): Move to make.
41          """
42          self._model.make_move(move)
43          self._view.set_overlay_coords([], None)
44
45          if self._model.states['CPU_ENABLED']:
46              self._model.make_cpu_move()
47
48      def handle_pause_event(self, event):
49          """
50          Processes events when game is paused.
51
52          Args:
53              event (GameEventType): Event to process.
54
55          Raises:
56              Exception: If event type is unrecognised.
57          """
58          game_event = self._pause_view.convert_mouse_pos(event)
59
60          if game_event is None:
61              return
62
63          match game_event.type:
64              case GameEventType.PAUSE_CLICK:
65                  self._model.toggle_paused()
66
67              case GameEventType.MENU_CLICK:
68                  self.cleanup('menu')
69
70              case _:
71                  raise Exception('Unhandled event type (GameController.handle_event
    )')
72
73      def handle_winner_event(self, event):
74          """
75          Processes events when game is over.
76
77          Args:
78              event (GameEventType): Event to process.
79
80          Raises:
81              Exception: If event type is unrecognised.
82          """
83          game_event = self._win_view.convert_mouse_pos(event)
84
85          if game_event is None:
86              return
```

```
87
88          match game_event.type:
89              case GameEventType.MENU_CLICK:
90                  self.cleanup('menu')
91                  return
92
93              case GameEventType.GAME_CLICK:
94                  self.cleanup('game')
95                  return
96
97              case _:
98                  raise Exception('Unhandled event type (GameController.handle_event
    )')
99
100     def handle_game_widget_event(self, event):
101         """
102         Processes events for game GUI widgets.
103
104         Args:
105             event (GameEventType): Event to process.
106
107         Raises:
108             Exception: If event type is unrecognised.
109
110         Returns:
111             CustomEvent | None: A widget event.
112         """
113         widget_event = self._view.process_widget_event(event)
114
115         if widget_event is None:
116             return None
117
118         match widget_event.type:
119             case GameEventType.ROTATE_PIECE:
120                 src_coords = self._view.get_selected_coords()
121
122                 if src_coords is None:
123                     logger.info('None square selected')
124                     return
125
126                 move = Move.instance_from_coords(MoveType.ROTATE, src_coords,
    src_coords, rotation_direction=widget_event.rotation_direction)
127                 self.make_move(move)
128
129             case GameEventType.RESIGN_CLICK:
130                 self._model.set_winner(self._model.states['ACTIVE_COLOUR'].
    get_flipped_colour())
131                 self._view.set_status_text(StatusText.WIN)
132
133             case GameEventType.DRAW_CLICK:
134                 self._model.set_winner(Miscellaneous.DRAW)
135                 self._view.set_status_text(StatusText.DRAW)
136
137             case GameEventType.TIMER_END:
138                 if self._model.states['TIME_ENABLED']:
139                     self._model.set_winner(widget_event.active_colour.
    get_flipped_colour())
140
141             case GameEventType.MENU_CLICK:
142                 self.cleanup('menu')
143
144             case GameEventType.HELP_CLICK:
```

```
145                 self._view.add_help_screen()
146 
147             case GameEventType.TUTORIAL_CLICK:
148                 self._view.add_tutorial_screen()
149 
150             case _:
151                 raise Exception('Unhandled event type (GameController.handle_event
    )')
152 
153         return widget_event.type
154 
155     def check_cpu(self):
156         """
157         Checks if CPU calculations are finished every frame.
158         """
159         if self._model.states['CPU_ENABLED'] and self._model.states['AWAITING_CPU'
    ] is False:
160             self._model.check_cpu()
161 
162     def handle_game_event(self, event):
163         """
164         Processes Pygame events for main game.
165 
166         Args:
167             event (pygame.Event): If event type is unrecognised.
168 
169         Raises:
170             Exception: If event type is unrecognised.
171         """
172         # Pass event for widgets to process
173         widget_event = self.handle_game_widget_event(event)
174 
175         if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
    KEYDOWN]:
176             if event.type != pygame.KEYDOWN:
177                 game_event = self._view.convert_mouse_pos(event)
178             else:
179                 game_event = None
180 
181             if game_event is None:
182                 if widget_event is None:
183                     if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
184                         # If user releases mouse click not on a widget
185                         self._view.remove_help_screen()
186                         self._view.remove_tutorial_screen()
187                     if event.type == pygame.MOUSEBUTTONUP:
188                         # If user releases mouse click on neither a widget or
    board
189                         self._view.set_overlay_coords(None, None)
190 
191                 return
192 
193             match game_event.type:
194                 case GameEventType.BOARD_CLICK:
195                     if self._model.states['AWAITING_CPU']:
196                         return
197 
198                     clicked_coords = game_event.coords
199                     clicked_bitboard = bb_helpers.coords_to_bitboard(
    clicked_coords)
200                     selected_coords = self._view.get_selected_coords()
201 
```

```
202                     if selected_coords:
203                         if clicked_coords == selected_coords:
204                             # If clicking on an already selected square, start
    dragging piece on that square
205                             self._view.set_dragged_piece(*self._model.
    get_piece_info(clicked_bitboard))
206                             return
207 
208                         selected_bitboard = bb_helpers.coords_to_bitboard(
    selected_coords)
209                         available_bitboard = self._model.get_available_moves(
    selected_bitboard)
210 
211                         if bb_helpers.is_occupied(clicked_bitboard,
    available_bitboard):
212                             # If the newly clicked square is not the same as the
    old one, and is an empty surrounding square, make a move
213                             move = Move.instance_from_coords(MoveType.MOVE,
    selected_coords, clicked_coords)
214                             self.make_move(move)
215                         else:
216                             # If the newly clicked square is not the same as the
    old one, but is an invalid square, unselect the currently selected square
217                             self._view.set_overlay_coords(None, None)
218 
219                     # Select hovered square if it is same as active colour
220                     elif self._model.is_selectable(clicked_bitboard):
221                         available_bitboard = self._model.get_available_moves(
    clicked_bitboard)
222                         self._view.set_overlay_coords(bb_helpers.
    bitboard_to_coords_list(available_bitboard), clicked_coords)
223                         self._view.set_dragged_piece(*self._model.get_piece_info(
    clicked_bitboard))
224 
225                 case GameEventType.PIECE_DROP:
226                     hovered_coords = game_event.coords
227 
228                     # if piece is dropped onto the board
229                     if hovered_coords:
230                         hovered_bitboard = bb_helpers.coords_to_bitboard(
    hovered_coords)
231                         selected_coords = self._view.get_selected_coords()
232                         selected_bitboard = bb_helpers.coords_to_bitboard(
    selected_coords)
233                         available_bitboard = self._model.get_available_moves(
    selected_bitboard)
234 
235                         if bb_helpers.is_occupied(hovered_bitboard,
    available_bitboard):
236                             # Make a move if mouse is hovered over an empty
    surrounding square
237                             move = Move.instance_from_coords(MoveType.MOVE,
    selected_coords, hovered_coords)
238                             self.make_move(move)
239 
240                     if game_event.remove_overlay:
241                         self._view.set_overlay_coords(None, None)
242 
243                     self._view.remove_dragged_piece()
244 
245                 case _:
```

```
                    raise Exception('Unhandled event type (GameController.handle_event)', game_event.type)

    def handle_event(self, event):
        """
        Passe a Pygame event to the correct handling function according to the game state.

        Args:
            event (pygame.Event): Event to process.
        """
        if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.MOUSEMOTION, pygame.KEYDOWN]:
            if self._model.states['PAUSED']:
                self.handle_pause_event(event)
            elif self._model.states['WINNER'] is not None:
                self.handle_winner_event(event)
            else:
                self.handle_game_event(event)

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                self._model.toggle_paused()
            elif event.key == pygame.K_l:
                logger.info('\nSTOPPING CPU')
                self._model._cpu_thread.stop_cpu() #temp
```

1.5.4 Board

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

board.py

```
from 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 = ''
        pieces = defaultdict(int)

        for rank in reversed(Rank):
```

```
25              for file in File:
26                  mask = 1 << (rank * 10 + file)
27                  blue_piece = self.bitboards.get_piece_on(mask, Colour.BLUE)
28                  red_piece = self.bitboards.get_piece_on(mask, Colour.RED)
29
30                  if blue_piece:
31                      pieces[blue_piece.value.upper()] += 1
32                      characters += f'{blue_piece.upper()}  '
33                  elif red_piece:
34                      pieces[red_piece.value] += 1
35                      characters += f'{red_piece}  '
36                  else:
37                      characters += '.  '
38
39              characters += '\n\n'
40
41          characters += str(dict(pieces))
42          characters += f'\nCURRENT PLAYER TO MOVE: {self.bitboards.active_colour.
    name}\n'
43          return characters
44
45      def get_piece_list(self):
46          """
47          Converts the board bitboards to a list of pieces.
48
49          Returns:
50              list: List of Pieces.
51          """
52          return self.bitboards.convert_to_piece_list()
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)
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_all_valid_squares(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         piece_bitboard = self.bitboards.combined_colour_bitboards[colour]
221         possible_moves = 0b0
222 
223         for square in bb_helpers.occupied_squares(piece_bitboard):
224             possible_moves |= self.get_valid_squares(square)
225 
226         return possible_moves
227 
228     def get_all_active_pieces(self):
229         """
230         Gets all active pieces for the current player.
231 
232         Returns:
233             int: The bitboard representation of all active pieces.
234         """
235         active_pieces = self.bitboards.combined_colour_bitboards[self.bitboards.
    active_colour]
236         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, self.
    bitboards.active_colour)
237         return active_pieces ^ sphinx_bitboard
238 
239     def fire_laser(self, remove_hash):
240         """
241         Fires the laser and removes hit pieces.
242 
243         Args:
244             remove_hash (bool): Whether to clear the hash list if a piece is hit.
245 
246         Returns:
247             Laser: The result of firing the laser.
248         """
249         laser = Laser(self.bitboards)
250 
251         if laser.hit_square_bitboard:
252             self.remove_piece(laser.hit_square_bitboard)
253 
254             if remove_hash:
255                 self.hash_list = [] # Remove all hashes for threefold repetition,
    as the position is impossible to be repeated after a piece is removed
256         return laser
257 
258     def generate_square_moves(self, src):
259         """
```

```
260         Generates all valid moves for a piece on a given square.
261 
262         Args:
263             src (int): The bitboard representation of the source square.
264 
265         Yields:
266             Move: A valid move for the piece.
267         """
268         for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
269             yield Move(MoveType.MOVE, src, dest)
270 
271     def generate_all_moves(self, colour):
272         """
273         Generates all valid moves for a given colour.
274 
275         Args:
276             colour (Colour): The colour of the pieces.
277 
278         Yields:
279             Move: A valid move for the active colour.
280         """
281         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
282         # Remove source squares for Sphinx pieces, as they cannot be moved
283         sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour]
    ^ sphinx_bitboard
284 
285         for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
286             # Generate movement moves
287             yield from self.generate_square_moves(square)
288 
289             # Generate rotational moves
290             for rotation_direction in RotationDirection:
291                 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.info('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
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(
    rotation_1, bitboard)
140         self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.clear_square
    (rotation_2, bitboard)
141 
142         self._hasher.apply_rotation_hash(bitboard, old_rotation)
143 
144     def clear_square(self, bitboard, colour):
145         """
146         Clears a square piece and rotation for a given colour.
147 
148         Args:
149             bitboard (int): The bitboard representation of the square.
150             colour (Colour): The colour to clear.
151         """
152         piece = self.get_piece_on(bitboard, colour)
153 
154         if piece is None:
155             return
156 
157         piece_bitboard = self.get_piece_bitboard(piece, colour)
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)
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 
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`

```
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                  new_score = self.search(board, depth - 1, stop_event)[0]
50
51                  if new_score > max_score:
52                      max_score = new_score
53                      best_move = move
54                  elif new_score == max_score:
55                      # If evaluated scores are equal, pick a random move
56                      choice([best_move, move])
57
58                  board.undo_move(move, laser_result)
59
60              return max_score, best_move
61
62          else:
63              min_score = Score.INFINITE
64
65              for move in board.generate_all_moves(Colour.RED):
66                  laser_result = board.apply_move(move)
67                  new_score = self.search(board, depth - 1, stop_event)[0]
68
69                  if new_score < min_score:
70                      min_score = new_score
71                      best_move = move
72                  elif new_score == min_score:
73                      choice([best_move, move])
74
75                  board.undo_move(move, laser_result)
76
77              return min_score, best_move
```

1.6.2 Alpha-beta Pruning

alpha_beta.py

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

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

    def initialise_stats(self):
        """
        Initialises the number of prunes to the statistics dictionary to be logged.
        """
        super().initialise_stats()
        self._stats['beta_prunes'] = 0
        self._stats['alpha_prunes'] = 0

    def find_move(self, board, stop_event):
        """
        Finds the best move for the current board state.

        Args:
            board (Board): The current board state.
            stop_event (threading.Event): Event used to kill search from an external class.
        """
        self.initialise_stats()
        best_score, best_move = self.search(board, self._max_depth, -Score.INFINITE, Score.INFINITE, stop_event)

        if self._verbose:
            self.print_stats(best_score, best_move)

        self._callback(best_move)

    def search(self, board, depth, alpha, beta, stop_event):
        """
        Recursively DFS through minimax tree while pruning branches using the alpha and beta bounds.

        Args:
            board (Board): The current board state.
            depth (int): The current search depth.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.
            stop_event (threading.Event): Event used to kill search from an external class.

        Returns:
            tuple[int, Move]: The best score and the best move found.
        """
        if (base_case := super().search(board, depth, stop_event)):
            return base_case

        best_move = None

        # Blue is the maximising player
```

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

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 search(self, board, depth, alpha, beta, stop_event):
10         """
```

```
        Searches transposition table for a cached move before running a full search if necessary.
        Caches the searched result.

        Args:
            board (Board): The current board state.
            depth (int): The current search depth.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.
            stop_event (threading.Event): Event used to kill search from an external class.

        Returns:
            tuple[int, Move]: The best score and the best move found.
        """
        hash = board.to_hash()
        score, move = self._table.get_entry(hash, depth, alpha, beta)

        if score is not None:
            self._stats['cache_hits'] += 1
            self._stats['nodes'] += 1

            return score, move
        else:
            # If board hash entry not found in cache, run a full search
            score, move = super().search(board, depth, alpha, beta, stop_event)
            self._table.insert_entry(score, move, hash, depth, alpha, beta)

            return score, move

class TTMinimaxCPU(TranspositionTableMixin, ABMinimaxCPU):
    def initialise_stats(self):
        """
        Initialises cache statistics to be logged.
        """
        super().initialise_stats()
        self._stats['cache_hits'] = 0

    def print_stats(self, score, move):
        """
        Logs the statistics for the search.

        Args:
            score (int): The best score found.
            move (Move): The best move found.
        """
        # Calculate number of cached entries retrieved as a percentage of all nodes
        self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] / self._stats['nodes'], 3)
        self._stats['cache_entries'] = len(self._table._table)
        super().print_stats(score, move)
```

1.6.4 Evaluator

evaluator.py

```
from data.utils.bitboard_helpers import pop_count, occupied_squares, bitboard_to_index
from data.states.game.components.psqt import PSQT, FLIP
from data.managers.logs import initialise_logger
```

```
4 from data.constants import Colour, Piece, Score
5
6 logger = initialise_logger(__name__)
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_pieces(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_pieces(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('\nPosition:', self.evaluate_position(board, Colour.BLUE),
     self.evaluate_position(board, Colour.RED))
39             logger.info('Mobility:', self.evaluate_mobility(board, Colour.BLUE),
    self.evaluate_mobility(board, Colour.RED))
40             logger.info('Safety:', self.evaluate_pharoah_safety(board, Colour.BLUE
    ), self.evaluate_pharoah_safety(board, Colour.RED))
41             logger.info('Overall score', blue_score - red_score)
42
43         if absolute and board.get_active_colour() == Colour.RED:
44             return red_score - blue_score
45         else:
46             return blue_score - red_score
47
48     def evaluate_pieces(self, board, colour):
49         """
50         Evaluates the material score for a given colour.
51
52         Args:
53             board (Board): The current board state.
54             colour (Colour): The colour to evaluate.
55
56         Returns:
57             int: Sum of all piece scores.
58         """
59         return (
60             Score.SPHINX * board.bitboards.get_piece_count(Piece.SPHINX, colour) +
```

```
61              Score.PYRAMID * board.bitboards.get_piece_count(Piece.PYRAMID, colour)
     +
62              Score.ANUBIS * board.bitboards.get_piece_count(Piece.ANUBIS, colour) +
63              Score.SCARAB * board.bitboards.get_piece_count(Piece.SCARAB, colour)
64          )
65
66      def evaluate_position(self, board, colour):
67          """
68          Evaluates the positional score for a given colour.
69
70          Args:
71              board (Board): The current board state.
72              colour (Colour): The colour to evaluate.
73
74          Returns:
75              int: Score representing positional advantage/disadvantage.
76          """
77          score = 0
78
79          for piece in Piece:
80              if piece == Piece.SPHINX:
81                  continue
82
83              piece_bitboard = board.bitboards.get_piece_bitboard(piece, colour)
84
85              for bitboard in occupied_squares(piece_bitboard):
86                  index = bitboard_to_index(bitboard)
87                  # Flip PSQT if using from blue player's perspective
88                  index = FLIP[index] if colour == Colour.BLUE else index
89
90                  score += PSQT[piece][index] * Score.POSITION
91
92          return score
93
94      def evaluate_mobility(self, board, colour):
95          """
96          Evaluates the mobility score for a given colour.
97
98          Args:
99              board (Board): The current board state.
100             colour (Colour): The colour to evaluate.
101
102         Returns:
103             int: Score on numerical representation of mobility.
104         """
105         number_of_moves = pop_count(board.get_all_valid_squares(colour))
106
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         pharoah_available_moves = pop_count(board.get_valid_squares(
    pharoah_bitboard, colour))
122         return (8 - pharoah_available_moves) * Score.PHAROAH_SAFETY
```

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

```
1  import threading
2  import time
3  from data.managers.logs import initialise_logger
4
5  logger = initialise_logger(__name__)
6
7  class CPUThread(threading.Thread):
8      def __init__(self, cpu, verbose=False):
9          super().__init__()
10         self._stop_event = threading.Event()
11         self._running = True
12         self._verbose = verbose
13         self.daemon = True
14
15         self._board = None
16         self._cpu = cpu
17
18     def kill_thread(self):
19         """
20         Kills the CPU and terminates the thread by stopping the run loop.
21         """
22         self.stop_cpu()
23         self._running = False
24
25     def stop_cpu(self):
26         """
27         Kills the CPU's move search.
28         """
29         self._stop_event.set()
30         self._board = None
31
32     def start_cpu(self, board):
33         """
34         Starts the CPU's move search.
35
36         Args:
37             board (Board): The current board state.
38         """
39         self._stop_event.clear()
40         self._board = board
41
42     def run(self):
43         """
44         Periodically checks if the board variable is set.
45         If it is, then starts CPU search.
46         """
47         while self._running:
48             if self._board and self._cpu:
49                 self._cpu.find_move(self._board, self._stop_event)
50                 self.stop_cpu()
```

```
51              else:
52                  time.sleep(1)
53                  if self._verbose:
54                      logger.debug(f'(CPUThread.run) Thread {threading.get_native_id
    ()} idling...')
```

1.6.6 Zobrist Hashing

zobrist_hasher.py

```
1  from random import randint
2  from data.utils.bitboard_helpers import bitboard_to_index
3  from data.constants import Piece, Colour, Rotation
4
5  # Initialise random values for each piece type on every square
6  # (5 x 2 colours) pieces + 4 rotations, for 80 squares
7  zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)]
8  # Hash for when the red player's move
9  red_move_hash = randint(0, 2 ** 64)
10
11 # Maps piece to the correct random value
12 piece_lookup = {
13     Colour.BLUE: {
14         piece: i for i, piece in enumerate(Piece)
15     },
16     Colour.RED: {
17         piece: i + 5 for i, piece in enumerate(Piece)
18     },
19 }
20
21 # Maps rotation to the correct random value
22 rotation_lookup = {
23     rotation: i + 10 for i, rotation in enumerate(Rotation)
24 }
25
26 class ZobristHasher:
27     def __init__(self):
28         self.hash = 0
29
30     def get_piece_hash(self, index, piece, colour):
31         """
32         Gets the random value for the piece type on the given square.
33
34         Args:
35             index (int): The index of the square.
36             piece (Piece): The piece on the square.
37             colour (Colour): The colour of the piece.
38
39         Returns:
40             int: A 64-bit value.
41         """
42         piece_index = piece_lookup[colour][piece]
43         return zobrist_table[index][piece_index]
44
45     def get_rotation_hash(self, index, rotation):
46         """
47         Gets the random value for theon the given square.
48
49         Args:
50             index (int): The index of the square.
51             rotation (Rotation): The rotation on the square.
```

```
            colour (Colour): The colour of the piece.

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

    def apply_piece_hash(self, bitboard, piece, colour):
        """
        Updates the Zobrist hash with a new piece.

        Args:
            bitboard (int): The bitboard representation of the square.
            piece (Piece): The piece on the square.
            colour (Colour): The colour of the piece.
        """
        index = bitboard_to_index(bitboard)
        piece_hash = self.get_piece_hash(index, piece, colour)
        self.hash ^= piece_hash

    def apply_rotation_hash(self, bitboard, rotation):
        """Updates the Zobrist hash with a new rotation.

        Args:
            bitboard (int): The bitboard representation of the square.
            rotation (Rotation): The rotation on the square.
        """
        index = bitboard_to_index(bitboard)
        rotation_hash = self.get_rotation_hash(index, rotation)
        self.hash ^= rotation_hash

    def apply_red_move_hash(self):
        """
        Applies the Zobrist hash for the red player's move.
        """
        self.hash ^= red_move_hash
```

1.6.7 Cache

transposition_table.py

```
from data.constants import TranspositionFlag

class TranspositionEntry:
    def __init__(self, score, move, flag, hash_key, depth):
        self.score = score
        self.move = move
        self.flag = flag
        self.hash_key = hash_key
        self.depth = depth

class TranspositionTable:
    def __init__(self, max_entries=50000):
        self._max_entries = max_entries
        self._table = dict()

    def calculate_entry_index(self, hash_key):
        """
        Gets the dictionary key for a given Zobrist hash.

```

```
        Args:
            hash_key (int): A Zobrist hash.

        Returns:
            str: Key for the given hash.
        """
        # return hash_key % self._max_entries
        return str(hash_key)

    def insert_entry(self, score, move, hash_key, depth, alpha, beta):
        """
        Inserts an entry into the transposition table.

        Args:
            score (int): The evaluation score.
            move (Move): The best move found.
            hash_key (int): The Zobrist hash key.
            depth (int): The depth of the search.
            alpha (int): The upper bound value.
            beta (int): The lower bound value.

        Raises:
            Exception: Invalid depth or score.
        """
        if depth == 0 or alpha < score < beta:
            flag = TranspositionFlag.EXACT
            score = score
        elif score <= alpha:
            flag = TranspositionFlag.UPPER
            score = alpha
        elif score >= beta:
            flag = TranspositionFlag.LOWER
            score = beta
        else:
            raise Exception('(TranspositionTable.insert_entry)')

        self._table[self.calculate_entry_index(hash_key)] = TranspositionEntry(score, move, flag, hash_key, depth)

        if len(self._table) > self._max_entries:
            # Removes the longest-existing entry to free up space for more up-to-date entries
            # Expression to remove leftmost item taken from https://docs.python.org/3/library/collections.html#ordereddict-objects
            (k := next(iter(self._table)), self._table.pop(k))

    def get_entry(self, hash_key, depth, alpha, beta):
        """
        Gets an entry from the transposition table.

        Args:
            hash_key (int): The Zobrist hash key.
            depth (int): The depth of the search.
            alpha (int): The alpha value for pruning.
            beta (int): The beta value for pruning.

        Returns:
            tuple[int, Move] | tuple[None, None]: The evaluation score and the best move found, if entry exists.
        """
        index = self.calculate_entry_index(hash_key)

```

```
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

Every state class calls their `startup` method when switched to, and `cleanup` when exited. 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
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, occlusion_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()
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'])
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)
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)
```

```
    cursor = connection.cursor()

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

    connection.commit()
    connection.close()

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

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

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

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

change_fen_string_column_name_23122024.py

```
import sqlite3
from pathlib import Path

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

def upgrade():
    """
    Upgrade function to rename fen_string column.
    """
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

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

    connection.commit()
    connection.close()

def downgrade():
    """
    Downgrade function to revert fen_string column renaming.
    """
```

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

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

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

1.8.2 DML

database_helpers.py

```
import sqlite3
from pathlib import Path
from datetime import datetime

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

def insert_into_games(game_entry):
    """
    Inserts a new row into games table.

    Args:
        game_entry (GameEntry): GameEntry object containing game information.
    """
    connection = sqlite3.connect(database_path, detect_types=sqlite3.
    PARSE_DECLTYPES)
    cursor = connection.cursor()

    # Datetime added for created_dt column
    game_entry = (*game_entry, datetime.now())

    cursor.execute('''
        INSERT INTO games (cpu_enabled, cpu_depth, winner, time_enabled, time,
    number_of_ply, moves, start_fen_string, final_fen_string, created_dt)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    ''', game_entry)

    connection.commit()
    connection.close()

def get_all_games():
    """
    Get all rows in games table.

    Returns:
        list[dict]: List of game entries represented as dictionaries.
    """
    connection = sqlite3.connect(database_path, detect_types=sqlite3.
    PARSE_DECLTYPES)
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()

    cursor.execute('''
        SELECT * FROM games
    ''')
```

```
42      games = cursor.fetchall()
43
44      connection.close()
45
46      return [dict(game) for game in games]
47
48  def delete_all_games():
49      """
50      Delete all rows in games table.
51      """
52      connection = sqlite3.connect(database_path)
53      cursor = connection.cursor()
54
55      cursor.execute('''
56          DELETE FROM games
57      ''')
58
59      connection.commit()
60      connection.close()
61
62  def delete_game(id):
63      """
64      Deletes specific row in games table using id attribute.
65
66      Args:
67          id (int): Primary key for row.
68      """
69      connection = sqlite3.connect(database_path)
70      cursor = connection.cursor()
71
72      cursor.execute('''
73          DELETE FROM games WHERE id = ?
74      ''', (id,))
75
76      connection.commit()
77      connection.close()
78
79  def get_ordered_games(column, ascend=True, start_row=1, end_row=10):
80      """
81      Get specific number of rows from games table ordered by a specific column(s).
82
83      Args:
84          column (_type_): Column to sort by.
85          ascend (bool, optional): Sort ascending or descending. Defaults to True.
86          start_row (int, optional): First row returned. Defaults to 1.
87          end_row (int, optional): Last row returned. Defaults to 10.
88
89      Raises:
90          ValueError: If ascend argument or column argument are invalid types.
91
92      Returns:
93          list[dict]: List of ordered game entries represented as dictionaries.
94      """
95      if not isinstance(ascend, bool) or not isinstance(column, str):
96          raise ValueError('(database_helpers.get_ordered_games) Invalid input
        arguments!')
97
98      connection = sqlite3.connect(database_path, detect_types=sqlite3.
        PARSE_DECLTYPES)
99      connection.row_factory = sqlite3.Row
100     cursor = connection.cursor()
101
```

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

1.9 Shaders

1.9.1 Shader Manager

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

to aid with syntax highlighting in the fragment shader classes.

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

shader.py

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

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

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

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

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

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

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

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

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

```

```
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]])

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

        Args:
            shader_type (ShaderType): The type of shader.
            texture (moderngl.Texture): The texture to render.
            output_fbo (moderngl.Framebuffer, optional): The output framebuffer. Defaults to None.
            program_type (ShaderType, optional): The program type. Defaults to None.
            use_image (bool, optional): Whether to use the image uniform. Defaults to True.
            **kwargs: Additional uniforms for the fragment shader.
        """
        fbo = output_fbo or self.framebuffers[shader_type]
        program = self._programs[program_type] if program_type else self._programs[shader_type]
        vao= self._vaos[program_type] if program_type else self._vaos[shader_type]

        fbo.use()
        texture.use(0)

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

        vao.render(mode=moderngl.TRIANGLE_STRIP)

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

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

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

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

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

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

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

```
164             self._shader_list.remove(shader_type)
165
166     def render_output(self):
167         """
168         Renders the final output to the screen.
169         """
170         # Render to the screen framebuffer
171         self._ctx.screen.use()
172
173         # Take the texture of the last framebuffer to be rendered to, and render
    that to the screen framebuffer
174         output_shader_type = self._shader_list[-1]
175         self.get_fbo_texture(output_shader_type).use(0)
176         self._programs[output_shader_type]['image'] = 0
177
178         self._vaos[output_shader_type].render(mode=moderngl.TRIANGLE_STRIP)
179
180     def get_fbo_texture(self, shader_type):
181         """
182         Gets the texture from the specified shader type's FBO.
183
184         Args:
185             shader_type (ShaderType): The type of shader.
186
187         Returns:
188             moderngl.Texture: The texture from the FBO.
189         """
190         return self.framebuffers[shader_type].color_attachments[0]
191
192     def calibrate_pygame_surface(self, pygame_surface):
193         """
194         Converts the Pygame window surface into an OpenGL texture.
195
196         Args:
197             pygame_surface (pygame.Surface): The finished Pygame surface.
198
199         Returns:
200             moderngl.Texture: The calibrated texture.
201         """
202         texture = self._ctx.texture(pygame_surface.size, 4)
203         texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
204         texture.swizzle = 'BGRA'
205         # Take the Pygame surface's pixel array and draw it to the new texture
206         texture.write(pygame_surface.get_view('1'))
207
208         # ShaderType._CALIBRATE has a VAO containing the pygame_quad_array
    coordinates, as Pygame uses different texture coordinates than ModernGL
    textures
209         self.render_to_fbo(ShaderType._CALIBRATE, texture)
210         return self.get_fbo_texture(ShaderType._CALIBRATE)
211
212     def draw(self, surface, arguments):
213         """
214         Draws the Pygame surface with shaders applied to the screen.
215
216         Args:
217             surface (pygame.Surface): The final Pygame surface.
218             arguments (dict): A dict of { ShaderType: Args } items, containing
    keyword arguments for every fragment shader.
219         """
220         self._ctx.viewport = (0, 0, *self._screen_size)
221         texture = self.calibrate_pygame_surface(surface)
```

```
222
223         for shader_type in self._shader_list:
224             self._shader_passes[shader_type].apply(texture, **arguments.get(
    shader_type, {}))
225             texture = self.get_fbo_texture(shader_type)
226
227         self.render_output()
228
229     def __del__(self):
230         """
231         Cleans up ModernGL resources when the ShaderManager object is deleted.
232         """
233         self.cleanup()
234
235     def cleanup(self):
236         """
237         Cleans up resources used by the ModernGL.
238         Probably unnecessary as the 'auto' garbage collection mode is used.
239         """
240         self._pygame_buffer.release()
241         self._opengl_buffer.release()
242         for program in self._programs:
243             self._programs[program].release()
244         for texture in self._textures:
245             self._textures[texture].release()
246         for vao in self._vaos:
247             self._vaos[vao].release()
248         for framebuffer in self.framebuffers:
249             self.framebuffers[framebuffer].release()
250
251     def handle_resize(self, new_screen_size):
252         """
253         Handles resizing of the screen.
254
255         Args:
256             new_screen_size (tuple[int, int]): The new screen size.
257         """
258         self._screen_size = new_screen_size
259
260         # Recreate all framebuffers to prevent scaling issues
261         for shader_type in self.framebuffers:
262             filter = self._textures[shader_type].filter[0]
263             self.create_framebuffer(shader_type, size=self._screen_size, filter=
    filter)
```

1.9.2 Bloom

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

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

Extracting bright colours

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

highlight_brightness.frag

```
# version 330 core

in vec2 uvs;
out vec4 f_colour;

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

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

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

Blur

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

blur.py

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

BLUR_ITERATIONS = 4

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

        shader_manager.create_framebuffer(ShaderType._BLUR)

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

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

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

        for _ in range(BLUR_ITERATIONS):
            # Apply horizontal blur
            self._shader_manager.render_to_fbo(
                ShaderType._BLUR,
                texture=self._shader_manager.get_fbo_texture("blurPong"),
                output_fbo=self._shader_manager.framebuffers["blurPing"],
                passes=5,
```

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

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

blur.frag

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

in vec2 uvs;
out vec4 f_colour;

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

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

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

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

Combining

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

```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)
        highlight_texture = self._shader_manager.get_fbo_texture(ShaderType._HIGHLIGHT_BRIGHTNESS)

        # Use colour as highlights
        for colour in highlight_colours:
            _HighlightColour(self._shader_manager).apply(texture, old_highlight=
```

```
    highlight_texture, colour=colour, intensity=colour_intensity)
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;
```

```
uniform float THRESHOLD=0.99;

void main() {
  float maxDistance = 1.0;

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

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

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

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

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

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

Lightmap

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

lightmap.frag

```
# version 330 core

#define PI 3.14159265

in vec2 uvs;
out vec4 f_colour;

uniform float softShadow=0.1;
uniform float resolution;
uniform float falloff;
uniform vec3 lightColour;
uniform vec2 angleClamp;
uniform sampler2D occlusionMap;
uniform sampler2D image;

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

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

```

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

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

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

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

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

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

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

  sum += inLight * 0.16;

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

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

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

Class

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

rays.py

```
from data.shaders.classes.lightmap import _Lightmap
from data.shaders.classes.blend import _Blend
from data.shaders.protocol import SMProtocol
```

```
4  from data.shaders.classes.crop import _Crop
5  from data.constants import ShaderType
6
7  class Rays:
8      def __init__(self, shader_manager: SMProtocol, lights):
9          self._shader_manager = shader_manager
10         self._lights = lights
11
12         # Load all necessary shaders
13         shader_manager.load_shader(ShaderType._LIGHTMAP)
14         shader_manager.load_shader(ShaderType._BLEND)
15         shader_manager.load_shader(ShaderType._CROP)
16         shader_manager.create_framebuffer(ShaderType.RAYS)
17
18     def apply(self, texture, occlusion=None):
19         """
20         Applies the light rays effect to a given texture.
21
22         Args:
23             texture (moderngl.Texture): The texture to apply the effect to.
24             occlusion (pygame.Surface, optional): A Pygame mask surface to use as
       the occlusion texture. Defaults to None.
25         """
26         final_texture = texture
27
28         # Iterate through array containing light information
29         for pos, radius, colour, *args in self._lights:
30             # Topleft of final light source
31             light_topleft = (pos[0] - (radius * texture.size[1] / texture.size[0])
       , pos[1] - radius)
32             # Relative size of light compared to texture
33             relative_size = (radius * 2 * texture.size[1] / texture.size[0],
       radius * 2)
34
35             # Crop texture to light source diameter, and to position light source
       at the center
36             _Crop(self._shader_manager).apply(texture, relative_pos=light_topleft,
        relative_size=relative_size)
37             cropped_texture = self._shader_manager.get_fbo_texture(ShaderType.
       _CROP)
38
39             if occlusion:
40                 # Calibrate Pygame mask surface and crop it
41                 occlusion_texture = self._shader_manager.calibrate_pygame_surface(
       occlusion)
42                 _Crop(self._shader_manager).apply(occlusion_texture, relative_pos=
       light_topleft, relative_size=relative_size)
43                 occlusion_texture = self._shader_manager.get_fbo_texture(
       ShaderType._CROP)
44             else:
45                 occlusion_texture = None
46
47             # Apply lightmap shader, shadowmap and occlusion are included within
       the _Lightmap class
48             _Lightmap(self._shader_manager).apply(cropped_texture, colour,
       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)
```

1.9.4 Stack