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.

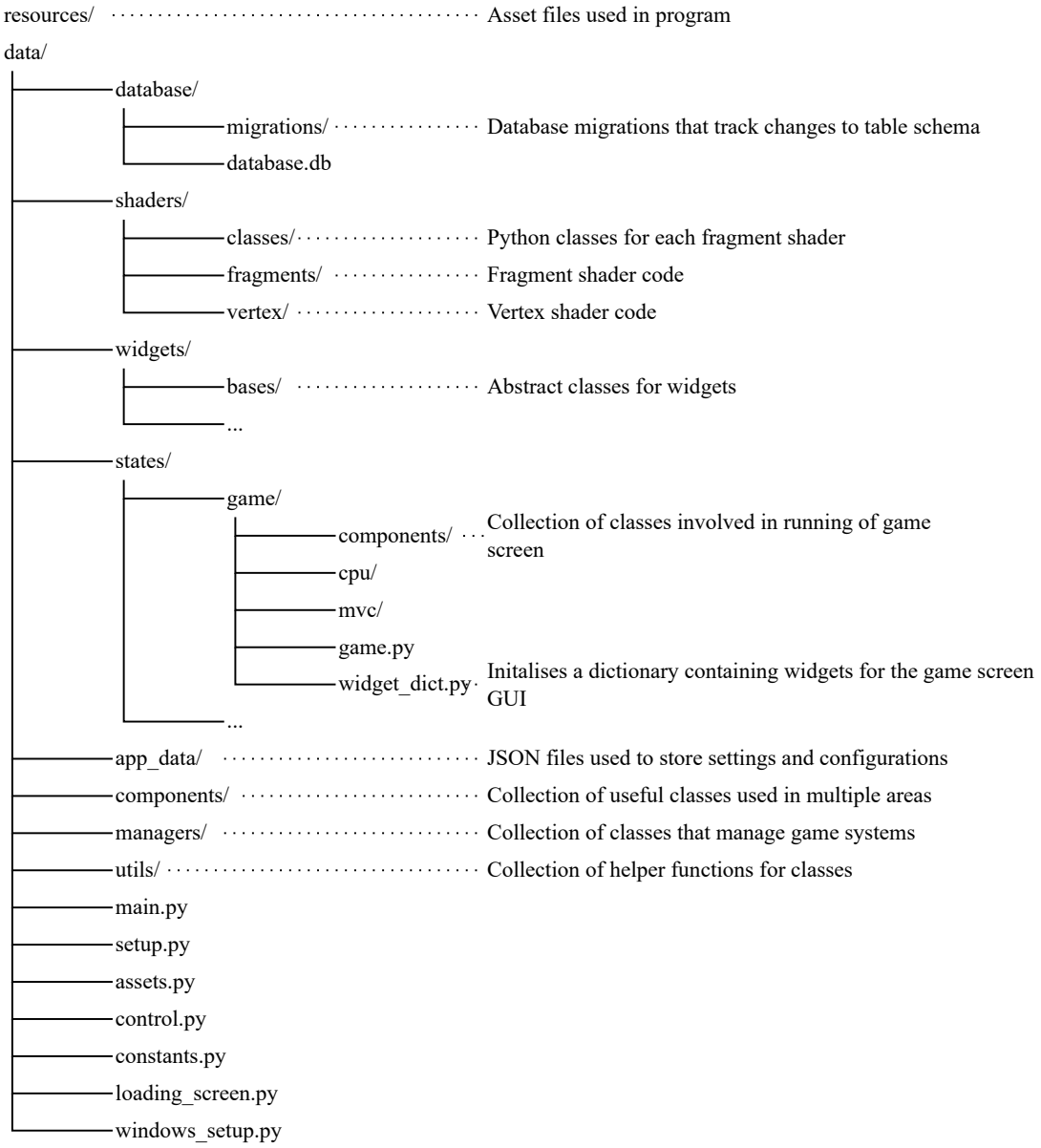

Figure 1: File tree diagram

1.2 Summary of Complexity

- Alpha-beta pruning and transposition table improvements for Minimax
- Shadow mapping and coordinate transformations
- Recursive Depth-First Search tree traversal (Theme)
- Circular doubly-linked list and stack

- Multipass shaders and gaussian blur
- Aggregate and Window SQL functions
- OOP techniques (Widget Bases and Widgets)
- Multithreading (Loading Screen)
- Bitboards
- (File handling and JSON parsing) (Helper functions)
- (Dictionary recursion)
- (Dot product) (Helper functions)

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
```

```
30
31 class LoadingScreen:
32     def __init__(self, target_func):
33         """
34         Creates new thread, and sets the load_state() function as its target.
35         Then starts draw loop for the loading screen.
36
37         Args:
38             target_func (Callable): function to be run on thread.
39         """
40         self._clock = pygame.time.Clock()
41         self._thread = threading.Thread(target=target_func)
42         self._thread.start()
43
44         self._logo_surface = load_gfx(logo_gfx_path)
45         self._logo_surface = pygame.transform.scale(self._logo_surface, (96, 96))
46         audio.play_sfx(load_sfx(sfx_path_1))
47         audio.play_sfx(load_sfx(sfx_path_2))
48
49         self.run()
50
51     @property
52     def logo_position(self):
53         duration = 1000
54         displacement = 50
55         elapsed_ticks = pygame.time.get_ticks() - start_ticks
56         progress = min(1, elapsed_ticks / duration)
57         center_pos = ((window.screen.size[0] - self._logo_surface.size[0]) / 2, (
    window.screen.size[1] - self._logo_surface.size[1]) / 2)
58
59         return (center_pos[0], center_pos[1] + displacement - displacement *
    easeOutBack(progress))
60
61     @property
62     def logo_opacity(self):
63         return min(255, (pygame.time.get_ticks() - start_ticks) / 5)
64
65     @property
66     def duration_not_over(self):
67         return (pygame.time.get_ticks() - start_ticks) < 1500
68
69     def event_loop(self):
70         """
71         Handles events for the loading screen, no user input is taken except to
    quit the game.
72         """
73         for event in pygame.event.get():
74             if event.type == pygame.QUIT:
75                 pygame.quit()
76                 sys.exit()
77
78     def draw(self):
79         """
80         Draws logo to screen.
81         """
82         window.screen.fill((0, 0, 0))
83
84         self._logo_surface.set_alpha(self.logo_opacity)
85         window.screen.blit(self._logo_surface, self.logo_position)
86
87         window.update()
88
```

```
    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.
    """
    return pygame.transform.smoothscale(image, target_size)

def gif_to_frames(path):
    """
    Uses the PIL library to break down GIFs into individual frames.

    Args:
        path (str): Directory path to GIF file.

    Yields:
        PIL.Image: Single frame.
    """
    try:
        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
```

```
    Returns:
        float: Angle between vectors.
    """
    dot_product = sum(i * j for (i, j) in zip(u, v))
    u_magnitude = math.sqrt(u[0] ** 2 + u[1] ** 2)
    v_magnitude = math.sqrt(v[0] ** 2 + v[1] ** 2)

    cos_angle = dot_product / (u_magnitude * v_magnitude)
    radians = math.acos(min(max(cos_angle, -1), 1))

    if deg:
        return math.degrees(radians)
    else:
        return radians

def get_rotational_angle(u, v, deg=True):
    """
    Get bearing angle relative to positive x-axis centered on second vector.

    Args:
        u (list[int, int]): Vector 1.
        v (list[int, int]): Vector 2, set as center of axes.
        deg (bool, optional): Return results in degrees. Defaults to True.

    Returns:
        float: Bearing angle between vectors.
    """
    radians = math.atan2(u[1] - v[1], u[0] -v[0])

    if deg:
        return math.degrees(radians)
    else:
        return radians

def get_vector(src_vertex, dest_vertex):
    """
    Get vector describing translation between two points.

    Args:
        src_vertex (list[int, int]): Source vertex.
        dest_vertex (list[int, int]): Destination vertex.

    Returns:
        tuple[int, int]: Vector between the two points.
    """
    return (dest_vertex[0] - src_vertex[0], dest_vertex[1] - src_vertex[1])

def get_next_corner(vertex, image_size):
    """
    Used in particle drawing system, finds coordinates of the next corner going
    clockwise, given a point on the perimeter.

    Args:
        vertex (list[int, int]): Point on perimeter.
        image_size (list[int, int]): Image size.

    Returns:
        list[int, int]: Coordinates of corner on perimeter.
    """
    corners = [(0, 0), (image_size[0], 0), (image_size[0], image_size[1]), (0,
    image_size[1])]

```

```
    if vertex in corners:
        return corners[(corners.index(vertex) + 1) % len(corners)]

    if vertex[1] == 0:
        return (image_size[0], 0)
    elif vertex[0] == image_size[0]:
        return image_size
    elif vertex[1] == image_size[1]:
        return (0, image_size[1])
    elif vertex[0] == 0:
        return (0, 0)

def pil_image_to_surface(pil_image):
    """
    Args:
        pil_image (PIL.Image): Image to be converted.

    Returns:
        pygame.Surface: Converted image surface.
    """
    return pygame.image.frombytes(pil_image.tobytes(), pil_image.size, pil_image.mode).convert()

def calculate_frame_index(elapsed_milliseconds, start_index, end_index, fps):
    """
    Determine frame of animated GIF to be displayed.

    Args:
        elapsed_milliseconds (int): Milliseconds since GIF started playing.
        start_index (int): Start frame of GIF.
        end_index (int): End frame of GIF.
        fps (int): Number of frames to be played per second.

    Returns:
        int: Displayed frame index of GIF.
    """
    ms_per_frame = int(1000 / fps)
    return start_index + ((elapsed_milliseconds // ms_per_frame) % (end_index - start_index))

def draw_background(screen, background, current_time=0):
    """
    Draws background to screen

    Args:
        screen (pygame.Surface): Screen to be drawn to
        background (list[pygame.Surface, ...] | pygame.Surface): Background to be drawn, if GIF, list of surfaces indexed to select frame to be drawn
        current_time (int, optional): Used to calculate frame index for GIF. Defaults to 0.
    """
    if isinstance(background, list):
        # Animated background passed in as list of surfaces, calculate_frame_index() used to get index of frame to be drawn
        frame_index = calculate_frame_index(current_time, 0, len(background), fps=8)
        scaled_background = scale_and_cache(background[frame_index], screen.size)
        screen.blit(scaled_background, (0, 0))
    else:
        scaled_background = scale_and_cache(background, screen.size)
        screen.blit(scaled_background, (0, 0))

```

```
def get_highlighted_icon(icon):
    """
    Used for pressable icons, draws overlay on icon to show as pressed.

    Args:
        icon (pygame.Surface): Icon surface.

    Returns:
        pygame.Surface: Icon with overlay drawn on top.
    """
    icon_copy = icon.copy()
    overlay = pygame.Surface((icon.get_width(), icon.get_height()), pygame.
    SRCALPHA)
    overlay.fill((0, 0, 0, 128))
    icon_copy.blit(overlay, (0, 0))
    return icon_copy
```

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

```
import json
from pathlib import Path

module_path = Path(__file__).parent
default_file_path = (module_path / '../app_data/default_settings.json').resolve()
user_file_path = (module_path / '../app_data/user_settings.json').resolve()
themes_file_path = (module_path / '../app_data/themes.json').resolve()

def load_json(path):
    """
    Args:
        path (str): Path to JSON file.

    Raises:
        Exception: Invalid file.

    Returns:
        dict: Parsed JSON file.
    """
    try:
        with open(path, 'r') as f:
            file = json.load(f)

        return file
    except:
        raise Exception('Invalid JSON file (data_helpers.py)')

def get_user_settings():
    return load_json(user_file_path)

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.
```

```
43
44      Raises:
45          Exception: Invalid file.
46      """
47      try:
48          with open(user_file_path, 'w') as f:
49              json.dump(data, f, indent=4)
50      except:
51          raise Exception('Invalid JSON file (data_helpers.py)')
```

widget_helpers.py (Files used for creating widgets)

```
1  import pygame
2  from math import sqrt
3
4  def create_slider(size, fill_colour, border_width, border_colour):
5      """
6      Creates surface for sliders.
7
8      Args:
9          size (list[int, int]): Image size.
10         fill_colour (pygame.Color): Fill (inner) colour.
11         border_width (float): Border width.
12         border_colour (pygame.Color): Border colour.
13
14     Returns:
15         pygame.Surface: Slider image surface.
16     """
17     gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
18     border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
       height))
19
20     # Draws rectangle with a border radius half of image height, to draw an
       rectangle with semicurclar cap (obround)
21     pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int
       (size[1] / 2))
22     pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
       border_width), border_radius=int(size[1] / 2))
23
24     return gradient_surface
25
26 def create_slider_gradient(size, border_width, border_colour):
27     """
28     Draws surface for colour slider, with a full colour gradient as fill colour.
29
30     Args:
31         size (list[int, int]): Image size.
32         border_width (float): Border width.
33         border_colour (pygame.Color): Border colour.
34
35     Returns:
36         pygame.Surface: Slider image surface.
37     """
38     gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
39
40     first_round_end = gradient_surface.height / 2
41     second_round_end = gradient_surface.width - first_round_end
42     gradient_y_mid = gradient_surface.height / 2
43
44     # Iterate through length of slider
45     for i in range(gradient_surface.width):
```

```
46          draw_height = gradient_surface.height
47
48          if i < first_round_end or i > second_round_end:
49              # Draw semicircular caps if x-distance less than or greater than
    radius of cap (half of image height)
50              distance_from_cutoff = min(abs(first_round_end - i), abs(i -
    second_round_end))
51              draw_height = calculate_gradient_slice_height(distance_from_cutoff,
    gradient_surface.height / 2)
52
53          # Get colour from distance from left side of slider
54          color = pygame.Color(0)
55          color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)
56
57          draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
58          draw_rect.center = (i, gradient_y_mid)
59
60          pygame.draw.rect(gradient_surface, color, draw_rect)
61
62      border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
    height))
63      pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
    border_width), border_radius=int(size[1] / 2))
64
65      return gradient_surface
66
67  def calculate_gradient_slice_height(distance, radius):
68      """
69      Calculate height of vertical slice of semicircular slider cap.
70
71      Args:
72          distance (float): x-distance from center of circle.
73          radius (float): Radius of semicircle.
74
75      Returns:
76          float: Height of vertical slice.
77      """
78      return sqrt(radius ** 2 - distance ** 2) * 2 + 2
79
80  def create_slider_thumb(radius, colour, border_colour, border_width):
81      """
82      Creates surface with bordered circle.
83
84      Args:
85          radius (float): Radius of circle.
86          colour (pygame.Color): Fill colour.
87          border_colour (pygame.Color): Border colour.
88          border_width (float): Border width.
89
90      Returns:
91          pygame.Surface: Circle surface.
92      """
93      thumb_surface = pygame.Surface((radius * 2, radius * 2), pygame.SRCALPHA)
94      pygame.draw.circle(thumb_surface, border_colour, (radius, radius), radius,
    width=int(border_width))
95      pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius -
    border_width))
96
97      return thumb_surface
98
99  def create_square_gradient(side_length, colour):
100     """
```

```
101     Creates a square gradient for the colour picker widget, gradient transitioning
         between saturation and value.
102     Uses smoothscale to blend between colour values for individual pixels.
103 
104     Args:
105         side_length (float): Length of a square side.
106         colour (pygame.Color): Colour with desired hue value.
107 
108     Returns:
109         pygame.Surface: Square gradient surface.
110     """
111     square_surface = pygame.Surface((side_length, side_length))
112 
113     mix_1 = pygame.Surface((1, 2))
114     mix_1.fill((255, 255, 255))
115     mix_1.set_at((0, 1), (0, 0, 0))
116     mix_1 = pygame.transform.smoothscale(mix_1, (side_length, side_length))
117 
118     hue = colour.hsva[0]
119     saturated_rgb = pygame.Color(0)
120     saturated_rgb.hsva = (hue, 100, 100)
121 
122     mix_2 = pygame.Surface((2, 1))
123     mix_2.fill((255, 255, 255))
124     mix_2.set_at((1, 0), saturated_rgb)
125     mix_2 = pygame.transform.smoothscale(mix_2,(side_length, side_length))
126 
127     mix_1.blit(mix_2, (0, 0), special_flags=pygame.BLEND_MULT)
128 
129     square_surface.blit(mix_1, (0, 0))
130 
131     return square_surface
132 
133 def create_switch(size, colour):
134     """
135     Creates surface for switch toggle widget.
136 
137     Args:
138         size (list[int, int]): Image size.
139         colour (pygame.Color): Fill colour.
140 
141     Returns:
142         pygame.Surface: Switch surface.
143     """
144     switch_surface = pygame.Surface((size[0], size[1]), pygame.SRCALPHA)
145     pygame.draw.rect(switch_surface, colour, (0, 0, size[0], size[1]),
     border_radius=int(size[1] / 2))
146 
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.
```

```
    """
    surface = pygame.Surface(size, pygame.SRCALPHA)

    pygame.draw.rect(surface, colours[0], (0, 0, *size))
    pygame.draw.rect(surface, colours[2], (border_width, border_width, size[0] - 2 * border_width, size[1] - 2 * border_width))
    pygame.draw.rect(surface, colours[3], (border_width, border_width, size[0] - 2 * border_width, border_width))
    pygame.draw.rect(surface, colours[1], (border_width, size[1] - 2 * border_width, size[0] - 2 * border_width, border_width))

    return surface
```

1.3.4 Theme

The theme manager file is responsible for providing an instance where the colour palette and dimensions for the GUI can be accessed.

theme.py

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

themes = get_themes()
user_settings = get_user_settings()

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

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

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

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

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

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

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

        Args:
            arg (str): Attribute name.
```

```
42
43          Raises:
44              KeyError: Instance does not have requested attribute.
45
46          Returns:
47              str | int: Instance attribute.
48          """
49          item = self.__dict__.get(arg)
50
51          if item is None:
52              raise KeyError('(ThemeManager.__getitem__) Requested theme item not
    found:', arg)
53
54          return item
55
56  theme = ThemeManager()
```

1.4 GUI

1.4.1 Laser

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

laser_draw.py

```
1   import pygame
2   from data.utils.board_helpers import coords_to_screen_pos
3   from data.constants import EMPTY_BB, ShaderType, Colour
4   from data.managers.animation import animation
5   from data.managers.window import window
6   from data.managers.audio import audio
7   from data.assets import GRAPHICS, SFX
8   from data.constants import LaserType
9
10  type_to_image = {
11      LaserType.END: ['laser_end_1', 'laser_end_2'],
12      LaserType.STRAIGHT: ['laser_straight_1', 'laser_straight_2'],
13      LaserType.CORNER: ['laser_corner_1', 'laser_corner_2']
14  }
15
16  GLOW_SCALE_FACTOR = 1.5
17
18  class LaserDraw:
19      def __init__(self, board_position, board_size):
20          self._board_position = board_position
21          self._square_size = board_size[0] / 10
22          self._laser_lists = []
23
24      @property
25      def firing(self):
26          return len(self._laser_lists) > 0
27
28      def add_laser(self, laser_result, laser_colour):
29          """
30          Adds a laser to the board.
31
32          Args:
33              laser_result (Laser): Laser class instance containing laser trajectory
     info.
34              laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
35          """
36          laser_path = laser_result.laser_path.copy()
```

```
37          laser_types = [LaserType.END]
38          # List of angles in degree to rotate the laser image surface when drawn
39          laser_rotation = [laser_path[0][1]]
40          laser_lights = []
41
42          # Iterates through every square laser passes through
43          for i in range(1, len(laser_path)):
44              previous_direction = laser_path[i-1][1]
45              current_coords, current_direction = laser_path[i]
46
47              if current_direction == previous_direction:
48                  laser_types.append(LaserType.STRAIGHT)
49                  laser_rotation.append(current_direction)
50              elif current_direction == previous_direction.get_clockwise():
51                  laser_types.append(LaserType.CORNER)
52                  laser_rotation.append(current_direction)
53              elif current_direction == previous_direction.get_anticlockwise():
54                  laser_types.append(LaserType.CORNER)
55                  laser_rotation.append(current_direction.get_anticlockwise())
56
57              # Adds a shader ray effect on the first and last square of the laser
    trajectory
58              if i in [1, len(laser_path) - 1]:
59                  abs_position = coords_to_screen_pos(current_coords, self.
    _board_position, self._square_size)
60                  laser_lights.append([
61                      (abs_position[0] / window.size[0], abs_position[1] / window.
    size[1]),
62                      0.5,
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

```
import pygame
from random import randint
from data.utils.asset_helpers import get_perimeter_sample, get_vector,
    get_angle_between_vectors, get_next_corner
from data.states.game.components.piece_sprite import PieceSprite

class ParticlesDraw:
    def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
        self._particles = []
        self._glow_particles = []

        self._gravity = gravity
        self._rotation = rotation
        self._shrink = shrink
        self._opacity = opacity

    def fragment_image(self, image, number):
        image_size = image.get_rect().size
        """
        1. Takes an image surface and samples random points on the perimeter.
        2. Iterates through points, and depending on the nature of two consecutive
     points, finds a corner between them.
        3. Draws a polygon with the points as the vertices to mask out the area
    not in the fragment.

        Args:
            image (pygame.Surface): Image to fragment.
            number (int): The number of fragments to create.

        Returns:
            list[pygame.Surface]: List of image surfaces with fragment of original
     surface drawn on top.
        """
        center = image.get_rect().center
        points_list = get_perimeter_sample(image_size, number)
        fragment_list = []

        points_list.append(points_list[0])

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

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

            corners_to_draw = None

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

```
                corners_to_draw = 4

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

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

            else:
                corners_to_draw = 1

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

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

            fragment_list.append(cropped_image)

        return fragment_list

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

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

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

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

    def add_sparks(self, radius, colour, position):
        """
        Adds laser spark particles.

        Args:
            radius (int): The radius of the sparks.
            colour (Colour.BLUE | Colour.RED): The active colour of the sparks.
            position (tuple[int, int]): The position where particles originate from.
        """
        for i in range(randint(10, 15)):
            velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
            random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in colour]
```

```
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)

                # Update opacity
                alpha = 255 - particle[4] * (255 - self._opacity)

                updated_image.fill((255, 255, 255, alpha), None, pygame.
        BLEND_RGBA_MULT)

                particle[0] = updated_image

        def draw(self, screen):
            """
            Draws the particles, indexing the surface and position attributes for each
         particle.

            Args:
                screen (pygame.Surface): The screen to draw on.
            """
            screen.blits([
                (particle[0], particle[2]) for particle in self._particles
            ])
```

1.4.3 Widget Bases

Widget bases are 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.

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

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

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

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

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

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

```
21
22          Remains constant, relative to initial screen size:
23          relative_font_size (float, optional): The relative font size of the widget
    .
24          relative_margin (float): The relative margin of the widget.
25          relative_border_width (float): The relative border width of the widget.
26          relative_border_radius (float): The relative border radius of the widget.
27
28          anchor_x (str): The horizontal anchor direction ('left', 'right', 'center
    ').
29          anchor_y (str): The vertical anchor direction ('top', 'bottom', 'center').
30          fixed_position (tuple[int, int], optional): The fixed position of the
    widget in pixels.
31          border_colour (pygame.Color): The border color of the widget.
32          text_colour (pygame.Color): The text color of the widget.
33          fill_colour (pygame.Color): The fill color of the widget.
34          font (pygame.freetype.Font): The font used for the widget.
35          """
36          super().__init__()
37
38          for required_kwarg in REQUIRED_KWARGS:
39              if required_kwarg not in kwargs:
40                  raise KeyError(f'(_Widget.__init__) Required keyword "{
    required_kwarg}" not in base kwargs')
41
42          self._surface = None # Set in WidgetGroup, as needs to be reassigned every
     frame
43          self._raw_surface_size = DEFAULT_SURFACE_SIZE
44
45          self._parent = kwargs.get('parent')
46
47          self._relative_font_size = None # Set in subclass
48
49          self._relative_position = kwargs.get('relative_position')
50          self._relative_margin = theme['margin'] / self._raw_surface_size[1]
51          self._relative_border_width = theme['borderWidth'] / self.
    _raw_surface_size[1]
52          self._relative_border_radius = theme['borderRadius'] / self.
    _raw_surface_size[1]
53
54          self._border_colour = pygame.Color(theme['borderPrimary'])
55          self._text_colour = pygame.Color(theme['textPrimary'])
56          self._fill_colour = pygame.Color(theme['fillPrimary'])
57          self._font = DEFAULT_FONT
58
59          self._anchor_x = kwargs.get('anchor_x') or 'left'
60          self._anchor_y = kwargs.get('anchor_y') or 'top'
61          self._fixed_position = kwargs.get('fixed_position')
62          scale_mode = kwargs.get('scale_mode') or 'both'
63
64          if kwargs.get('relative_size'):
65              match scale_mode:
66                  case 'height':
67                      self._relative_size = kwargs.get('relative_size')
68                  case 'width':
69                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] *
    self.surface_size[0]) / self.surface_size[1])
70                  case 'both':
71                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
72                  case _:
```

```
73                      raise ValueError('(_Widget.__init__) Unknown scale mode:',
    scale_mode)
74          else:
75              self._relative_size = (1, 1)
76
77          if 'margin' in kwargs:
78              self._relative_margin = kwargs.get('margin') / self._raw_surface_size
    [1]
79
80              if (self._relative_margin * 2) > min(self._relative_size[0], self.
    _relative_size[1]):
81                  raise ValueError('(_Widget.__init__) Margin larger than specified
    size!')
82
83          if 'border_width' in kwargs:
84              self._relative_border_width = kwargs.get('border_width') / self.
    _raw_surface_size[1]
85
86          if 'border_radius' in kwargs:
87              self._relative_border_radius = kwargs.get('border_radius') / self.
    _raw_surface_size[1]
88
89          if 'border_colour' in kwargs:
90              self._border_colour = pygame.Color(kwargs.get('border_colour'))
91
92          if 'fill_colour' in kwargs:
93              self._fill_colour = pygame.Color(kwargs.get('fill_colour'))
94
95          if 'text_colour' in kwargs:
96              self._text_colour = pygame.Color(kwargs.get('text_colour'))
97
98          if 'font' in kwargs:
99              self._font = kwargs.get('font')
100
101     @property
102     def surface_size(self):
103         """
104         Gets the size of the surface widget is drawn on.
105         Can be either the window size, or another widget size if assigned to a
    parent.
106
107         Returns:
108             tuple[int, int]: The size of the surface.
109         """
110         if self._parent:
111             return self._parent.size
112         else:
113             return self._raw_surface_size
114
115     @property
116     def position(self):
117         """
118         Gets the position of the widget.
119         Accounts for fixed position attribute, where widget is positioned in
    pixels regardless of screen size.
120         Acounts for anchor direction, where position attribute is calculated
    relative to one side of the screen.
121
122         Returns:
123             tuple[int, int]: The position of the widget.
124         """
125         x, y = None, None
```

```
126         if self._fixed_position:
127             x, y = self._fixed_position
128         if x is None:
129             x = self._relative_position[0] * self.surface_size[0]
130         if y is None:
131             y = self._relative_position[1] * self.surface_size[1]
132 
133         if self._anchor_x == 'left':
134             x = x
135         elif self._anchor_x == 'right':
136             x = self.surface_size[0] - x - self.size[0]
137         elif self._anchor_x == 'center':
138             x = (self.surface_size[0] / 2 - self.size[0] / 2) + x
139 
140         if self._anchor_y == 'top':
141             y = y
142         elif self._anchor_y == 'bottom':
143             y = self.surface_size[1] - y - self.size[1]
144         elif self._anchor_y == 'center':
145             y = (self.surface_size[1] / 2 - self.size[1] / 2) + y
146 
147         # Position widget relative to parent, if exists.
148         if self._parent:
149             return (x + self._parent.position[0], y + self._parent.position[1])
150         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':
```

```
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'right':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'center':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position

    def set_surface_size(self, new_surface_size):
        """
        Sets the new size of the surface widget is drawn on.

        Args:
            new_surface_size (tuple[int, int]): The new size of the surface.
        """
        self._raw_surface_size = new_surface_size

    def process_event(self, event):
        """
        Abstract method to handle events.

        Args:
            event (pygame.Event): The event to process.
        """
        raise NotImplementedError
```

The `Circular` class provides functionality to support widgets which rotate between text/icons.

circular.py

```
from data.components.circular_linked_list import CircularLinkedList

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

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

    @property
```

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

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

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

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

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

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

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

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

            self.set_next_item()
```

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):
        """
        Initializes a CircularLinkedList object.

        Args:
            list_to_convert (list, optional): Creates a linked list from existing items. Defaults to None.
```

```
14          """
15          self._head = None
16
17          if list_to_convert:
18              for item in list_to_convert:
19                  self.insert_at_end(item)
20
21      def __str__(self):
22          """
23          Returns a string representation of the circular linked list.
24
25          Returns:
26              str: Linked list formatted as string.
27          """
28          if self._head is None:
29              return '| empty |'
30
31          characters = '| -> '
32          current_node = self._head
33          while True:
34              characters += str(current_node.data) + ' -> '
35              current_node = current_node.next
36
37              if current_node == self._head:
38                  characters += '|'
39                  return characters
40
41      def insert_at_beginning(self, data):
42          """
43          Inserts a node at the beginning of the circular linked list.
44
45          Args:
46              data: The data to insert.
47          """
48          new_node = Node(data)
49
50          if self._head is None:
51              self._head = new_node
52              new_node.next = self._head
53              new_node.previous = self._head
54          else:
55              new_node.next = self._head
56              new_node.previous = self._head.previous
57              self._head.previous.next = new_node
58              self._head.previous = new_node
59
60              self._head = new_node
61
62      def insert_at_end(self, data):
63          """
64          Inserts a node at the end of the circular linked list.
65
66          Args:
67              data: The data to insert.
68          """
69          new_node = Node(data)
70
71          if self._head is None:
72              self._head = new_node
73              new_node.next = self._head
74              new_node.previous = self._head
75          else:
```

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

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

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

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

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

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

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

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

        Args:
            data: The data to delete.

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

        current_node = self._head

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

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

```

```
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.

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

custom_event.py

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

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

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

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

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

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

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

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

```
55
56              for kwarg in kwargs:
57                  if kwarg not in required_args[event_type]:
58                      raise ValueError(f"Argument '{kwarg}' not included in
    required_args dictionary for event '{event_type}'! (GameEvent.create_event)")
59
60              return event_cls(event_type, **kwargs)
61
62          else:
63              return event_cls(event_type)
```

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)

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

reactive_icon_button.py

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

```
                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
        )
```

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)
        for item in self._items_dict.values():
            item.set_surface_size(new_surface_size)

    def process_event(self, event):
        """
        Processes Pygame events.

        Args:
            event (pygame.Event): Event to process.

        Returns:
            CustomEvent: CustomEvent of current item, with current key included
        """
        widget_event = super().process_event(event)
        self.current_item.process_event(event)

        if widget_event:
            return CustomEvent(**vars(widget_event), data=self.current_key)
```

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

```
import pygame
from data.utils.widget_helpers import create_slider_gradient
```

```
from data.utils.asset_helpers import smoothscale_and_cache
from data.widgets.slider_thumb import _SliderThumb
from data.widgets.bases.widget import _Widget
from data.constants import WidgetState

class _ColourSlider(_Widget):
    def __init__(self, relative_width, **kwargs):
        super().__init__(relative_size=(relative_width, relative_width * 0.2), **kwargs)

        # Initialise slider thumb.
        self._thumb = _SliderThumb(radius=self.size[1] / 2, border_colour=self._border_colour)

        self._selected_percent = 0
        self._last_mouse_x = None

        self._gradient_surface = create_slider_gradient(self.gradient_size, self.border_width, self._border_colour)
        self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)

    @property
    def gradient_size(self):
        return (self.size[0] - 2 * (self.size[1] / 2), self.size[1] / 2)

    @property
    def gradient_position(self):
        return (self.size[1] / 2, self.size[1] / 4)

    @property
    def thumb_position(self):
        return (self.gradient_size[0] * self._selected_percent, 0)

    @property
    def selected_colour(self):
        colour = pygame.Color(0)
        colour.hsva = (int(self._selected_percent * 360), 100, 100, 100)
        return colour

    def calculate_gradient_percent(self, mouse_pos):
        """
        Calculate what percentage slider thumb is at based on change in mouse position.

        Args:
            mouse_pos (list[int, int]): Position of mouse on window screen.

        Returns:
            float: Slider scroll percentage.
        """
        if self._last_mouse_x is None:
            return

        x_change = (mouse_pos[0] - self._last_mouse_x) / (self.gradient_size[0] - 2 * self.border_width)
        return max(0, min(self._selected_percent + x_change, 1))

    def relative_to_global_position(self, position):
        """
        Transforms position from being relative to widget rect, to window screen.

        Args:
```

```
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
```

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)

    @property
    def text(self):
        if self.is_placeholder:
            return ''

        return self._text

    def relative_x_to_cursor_index(self, relative_x):
        """
        Calculates cursor index using mouse position relative to the widget position.

        Args:
            relative_x (int): Horizontal distance of the mouse from the left side of the widget.

        Returns:
            int: Cursor index.
        """
        current_width = 0

        for index, metrics in enumerate(self._font.get_metrics(self._text, size=self.font_size)):
            glyph_width = metrics[4]

            if current_width >= relative_x:
                return index

            current_width += glyph_width

        return len(self._text)

    def set_cursor_index(self, mouse_pos):
        """
        Sets cursor index based on mouse position.

        Args:
            mouse_pos (list[int, int]): Mouse position relative to window screen.
        """
        if mouse_pos is None:
            self._cursor_index = mouse_pos
            return

        relative_x = mouse_pos[0] - (self.margin / 2) - self.rect.left
        relative_x = max(0, relative_x)
        self._cursor_index = self.relative_x_to_cursor_index(relative_x)

    def focus_input(self, mouse_pos):
        """
        Draws cursor and sets cursor index when user clicks on widget.

        Args:
            mouse_pos (list[int, int]): Mouse position relative to window screen.
        """
        if self.is_placeholder:
            self._text = ''
            self.is_placeholder = False

        self.set_cursor_index(mouse_pos)
        self.set_image()
```

```
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:
236                     case pygame.K_BACKSPACE:
237                         if self._cursor_index > 0:
238                             self._text = self._text[:self._cursor_index - 1] +
    self._text[self._cursor_index:]
239                         self._cursor_index = max(0, self._cursor_index - 1)
240 
241                     case pygame.K_RIGHT:
242                         self._cursor_index = min(len(self._text), self.
    _cursor_index + 1)
243 
244                     case pygame.K_LEFT:
245                         self._cursor_index = max(0, self._cursor_index - 1)
246 
247                     case pygame.K_ESCAPE:
248                         self.unfocus_input()
249                         return CustomEvent(**vars(self._event), text=self.text)
250 
251                     case pygame.K_RETURN:
252                         self.unfocus_input()
253                         return CustomEvent(**vars(self._event), text=self.text)
254 
255                     case _:
256                         if not event.unicode:
257                             return
258 
259                         potential_text = self._text[:self._cursor_index] + event.
    unicode + self._text[self._cursor_index:]
260 
261                         # Validator lambda function used to check if inputted text
     is valid before displaying
```

```
                        # e.g. Time control input has a validator function
    checking if text represents a float
                        if self._validator(potential_text) is False:
                            return

                        self._text = potential_text
                        self._cursor_index += 1

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

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

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

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

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

        if self._cursor_index is not None:
            scaled_cursor = pygame.transform.scale(self._empty_cursor, self.
    cursor_size)
            scaled_cursor.fill(self._cursor_colour)
            self.image.blit(scaled_cursor, self.cursor_position)

    def update(self):
        """
        Overrides based update method, to handle cursor blinking.
        """
        super().update()
        # Calculate if cursor should be shown or not
        cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
        if cursor_frame == 1 and self._blinking_cooldown == 0:
            self._cursor_colour = (0, 0, 0, 0)
        else:
            self._cursor_colour = self._cursor_colour_copy
        self.set_image()
```

1.5 Game

1.5.1 Model

game_model.py

```
from data.states.game.components.fen_parser import encode_fen_string
from data.constants import Colour, GameEventType, EMPTY_BB
from data.states.game.widget_dict import GAME_WIDGETS
from data.states.game.cpu.cpu_thread import CPUThread
```

```
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

```python
import pygame
from data.constants import GameEventType, Colour, StatusText, Miscellaneous,
    ShaderType
from data.states.game.components.overlay_draw import OverlayDraw
from data.states.game.components.capture_draw import CaptureDraw
from data.states.game.components.piece_group import PieceGroup
from data.states.game.components.laser_draw import LaserDraw
from data.states.game.components.father import DragAndDrop
from data.utils.bitboard_helpers import bitboard_to_coords
from data.utils.board_helpers import screen_pos_to_coords
from data.states.game.widget_dict import GAME_WIDGETS
from data.components.custom_event import CustomEvent
from data.components.widget_group import WidgetGroup
from data.components.cursor import Cursor
from data.managers.window import window
from data.managers.audio import audio
from data.assets import SFX

class GameView:
    def __init__(self, model):
        self._model = model
        self._hide_pieces = False
        self._selected_coords = None
        self._event_to_func_map = {
            GameEventType.UPDATE_PIECES: self.handle_update_pieces,
            GameEventType.SET_LASER: self.handle_set_laser,
            GameEventType.PAUSE_CLICK: self.handle_pause,
        }

        # Register model event handling with process_model_event()
        self._model.register_listener(self.process_model_event, 'game')

        # Initialise WidgetGroup with map of widgets
        self._widget_group = WidgetGroup(GAME_WIDGETS)
        self._widget_group.handle_resize(window.size)
        self.initialise_widgets()

        self._cursor = Cursor()
        self._laser_draw = LaserDraw(self.board_position, self.board_size)
        self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
        self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
        self._capture_draw = CaptureDraw(self.board_position, self.board_size)
        self._piece_group = PieceGroup()
        self.handle_update_pieces()

        self.set_status_text(StatusText.PLAYER_MOVE)

    @property
    def board_position(self):
        return GAME_WIDGETS['chessboard'].position

    @property
    def board_size(self):
        return GAME_WIDGETS['chessboard'].size

    @property
    def square_size(self):
        return self.board_size[0] / 10

    def initialise_widgets(self):
        """
        Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
```

```
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          Handle resizing GUI.
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.BLUE | Colour.RED): 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

```
import pygame
from data.constants import GameEventType, MoveType, StatusText, Miscellaneous
from data.utils import bitboard_helpers as bb_helpers
from data.states.game.components.move import Move
from data.managers.logs import initialise_logger

logger = initialise_logger(__name__)

class GameController:
    def __init__(self, model, view, win_view, pause_view, to_menu, to_new_game):
        self._model = model
        self._view = view
        self._win_view = win_view
        self._pause_view = pause_view

        self._to_menu = to_menu
        self._to_new_game = to_new_game

        self._view.initialise_timers()

    def cleanup(self, next):
        """
        Handles game quit, either leaving to main menu or restarting a new game.

        Args:
            next (str): New state to switch to.
        """
        self._model.kill_thread()

        if next == 'menu':
            self._to_menu()
        elif next == 'game':
            self._to_new_game()

    def make_move(self, move):
        """
        Handles player move.

        Args:
            move (Move): Move to make.
        """
        self._model.make_move(move)
        self._view.set_overlay_coords([], None)

        if self._model.states['CPU_ENABLED']:
            self._model.make_cpu_move()

    def handle_pause_event(self, event):
        """
        Processes events when game is paused.

        Args:
            event (GameEventType): Event to process.

        Raises:
            Exception: If event type is unrecognised.
        """
        game_event = self._pause_view.convert_mouse_pos(event)
```

```
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 _:
246                     raise Exception('Unhandled event type (GameController.
    handle_event)', game_event.type)
247 
248     def handle_event(self, event):
249         """
250         Passe a Pygame event to the correct handling function according to the
    game state.
251 
252         Args:
253             event (pygame.Event): Event to process.
254         """
255         if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
    MOUSEMOTION, pygame.KEYDOWN]:
256             if self._model.states['PAUSED']:
257                 self.handle_pause_event(event)
258             elif self._model.states['WINNER'] is not None:
259                 self.handle_winner_event(event)
260             else:
261                 self.handle_game_event(event)
262 
263         if event.type == pygame.KEYDOWN:
264             if event.key == pygame.K_ESCAPE:
265                 self._model.toggle_paused()
266             elif event.key == pygame.K_l:
267                 logger.info('\nSTOPPING CPU')
268                 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, TEST_MASK
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):
        characters = ''
        pieces = defaultdict(int)

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

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

            characters += '\n\n'

        characters += str(dict(pieces))
        characters += f'\nCURRENT PLAYER TO MOVE: {self.bitboards.active_colour.name}\n'
        return characters

    def get_piece_list(self):
        return self.bitboards.convert_to_piece_list()

    def get_active_colour(self):
        return self.bitboards.active_colour

    def to_hash(self):
        return self.bitboards.get_hash()

    def check_win(self):
        for colour in Colour:
            if self.bitboards.get_piece_bitboard(Piece.PHAROAH, colour) == EMPTY_BB:
                # print('\n(Board.check_win) Returning', colour.get_flipped_colour
```

```
().name)
52                  return colour.get_flipped_colour()
53
54          if self.hash_list.count(self.hash_list[-1]) >= 3: # ONLY CHECKING LAST AS
    check_win() CALLED EVERY MOVE
55              return Miscellaneous.DRAW
56
57          return None
58
59      def apply_move(self, move, fire_laser=True, add_hash=False):
60          piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
    active_colour)
61
62          if piece_symbol is None:
63              raise ValueError('Invalid move - no piece found on source square')
64          elif piece_symbol == Piece.SPHINX:
65              raise ValueError('Invalid move - sphinx piece is immovable')
66
67          if move.move_type == MoveType.MOVE:
68              possible_moves = self.get_valid_squares(move.src)
69              if bb_helpers.is_occupied(move.dest, possible_moves) is False:
70                  raise ValueError('Invalid move - destination square is occupied')
71
72              piece_rotation = self.bitboards.get_rotation_on(move.src)
73
74              self.bitboards.update_move(move.src, move.dest)
75              self.bitboards.update_rotation(move.src, move.dest, piece_rotation)
76
77          elif move.move_type == MoveType.ROTATE:
78              piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
    active_colour)
79              piece_rotation = self.bitboards.get_rotation_on(move.src)
80
81              if move.rotation_direction == RotationDirection.CLOCKWISE:
82                  new_rotation = piece_rotation.get_clockwise()
83              elif move.rotation_direction == RotationDirection.ANTICLOCKWISE:
84                  new_rotation = piece_rotation.get_anticlockwise()
85
86              self.bitboards.update_rotation(move.src, move.src, new_rotation)
87
88          laser = None
89          if fire_laser:
90              laser = self.fire_laser(add_hash)
91
92          if add_hash:
93              self.hash_list.append(self.bitboards.get_hash())
94
95          self.bitboards.flip_colour()
96
97          return laser
98
99      def undo_move(self, move, laser_result):
100         self.bitboards.flip_colour()
101
102         if laser_result.hit_square_bitboard:
103             src = laser_result.hit_square_bitboard
104             piece = laser_result.piece_hit
105             colour = laser_result.piece_colour
106             rotation = laser_result.piece_rotation
107
108             self.bitboards.set_square(src, piece, colour)
109             self.bitboards.clear_rotation(src)
```

```
110             self.bitboards.set_rotation(src, rotation)
111 
112         if move.move_type == MoveType.MOVE:
113             reversed_move = Move.instance_from_bitboards(MoveType.MOVE, move.dest,
     move.src)
114         elif move.move_type == MoveType.ROTATE:
115             reversed_move = Move.instance_from_bitboards(MoveType.ROTATE, move.src
    , move.src, move.rotation_direction.get_opposite())
116 
117         self.apply_move(reversed_move, fire_laser=False)
118         self.bitboards.flip_colour()
119 
120     def remove_piece(self, square_bitboard):
121         self.bitboards.clear_square(square_bitboard, Colour.BLUE)
122         self.bitboards.clear_square(square_bitboard, Colour.RED)
123         self.bitboards.clear_rotation(square_bitboard)
124 
125     def get_valid_squares(self, src_bitboard, colour=None):
126         target_top_left = (src_bitboard & A_FILE_MASK & EIGHT_RANK_MASK) << 9
127         target_top_middle = (src_bitboard & EIGHT_RANK_MASK) << 10
128         target_top_right = (src_bitboard & J_FILE_MASK & EIGHT_RANK_MASK) << 11
129         target_middle_right = (src_bitboard & J_FILE_MASK) << 1
130 
131         target_bottom_right = (src_bitboard & J_FILE_MASK & ONE_RANK_MASK) >> 9
132         target_bottom_middle = (src_bitboard & ONE_RANK_MASK) >> 10
133         target_bottom_left = (src_bitboard & A_FILE_MASK & ONE_RANK_MASK)>> 11
134         target_middle_left = (src_bitboard & A_FILE_MASK) >> 1
135 
136         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
137 
138         if colour is not None:
139             valid_possible_moves = possible_moves & ~self.bitboards.
    combined_colour_bitboards[colour]
140         else:
141             valid_possible_moves = possible_moves & ~self.bitboards.
    combined_all_bitboard
142 
143         # valid_possible_moves = valid_possible_moves & TEST_MASK
144 
145         return valid_possible_moves
146 
147     def get_all_valid_squares(self, colour):
148         piece_bitboard = self.bitboards.combined_colour_bitboards[colour]
149         possible_moves = 0b0
150 
151         for square in bb_helpers.occupied_squares(piece_bitboard):
152             possible_moves |= self.get_valid_squares(square)
153 
154         return possible_moves
155 
156     def get_all_active_pieces(self):
157         active_pieces = self.bitboards.combined_colour_bitboards[self.bitboards.
    active_colour]
158         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, self.
    bitboards.active_colour)
159         return active_pieces ^ sphinx_bitboard
160 
161     def fire_laser(self, remove_hash):
162         laser = Laser(self.bitboards)
163 
```

```
164         if laser.hit_square_bitboard:
165             self.remove_piece(laser.hit_square_bitboard)
166 
167             if remove_hash:
168                 self.hash_list = [] # AS POSITION IMPOSSIBLE TO REPEAT
169         return laser
170 
171     def generate_square_moves(self, src):
172         for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
173             yield Move(MoveType.MOVE, src, dest)
174 
175     def generate_all_moves(self, colour):
176         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
177         sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour]
    ^ sphinx_bitboard
178 
179         for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
180             # yield from self.generate_square_moves(square)
181 
182             for rotation_direction in RotationDirection:
183                 yield Move(MoveType.ROTATE, square, rotation_direction=
    rotation_direction)
```

1.5.5 Bitboards

The `BitboardCollection` class uses helper functinos 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.utils import bitboard_helpers as bb_helpers
4  from data.states.game.cpu.zobrist_hasher import ZobristHasher
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         characters = ''
28         for rank in reversed(Rank):
```

```
29              for file in File:
30                  bitboard = 1 << (rank * 10 + file)
31
32                  colour = self.get_colour_on(bitboard)
33                  piece = self.get_piece_on(bitboard, Colour.BLUE) or self.
    get_piece_on(bitboard, Colour.RED)
34
35                  if piece is not None:
36                          characters += f'{piece.upper() if colour == Colour.BLUE
    else piece}  '
37                  else:
38                      characters += '.  '
39
40              characters += '\n\n'
41
42          return characters
43
44      def get_rotation_string(self):
45          characters = ''
46          for rank in reversed(Rank):
47
48              for file in File:
49                  mask = 1 << (rank * 10 + file)
50                  rotation = self.get_rotation_on(mask)
51                  has_piece = bb_helpers.is_occupied(self.combined_all_bitboard,
    mask)
52
53                  if has_piece:
54                      characters += f'{rotation.upper()}  '
55                  else:
56                      characters += '.  '
57
58              characters += '\n\n'
59
60          return characters
61
62      def initialise_hash(self):
63          for piece in Piece:
64              for colour in Colour:
65                  piece_bitboard = self.get_piece_bitboard(piece, colour)
66
67                  for occupied_bitboard in bb_helpers.occupied_squares(
    piece_bitboard):
68                      self._hasher.apply_piece_hash(occupied_bitboard, piece, colour
    )
69
70          for bitboard in bb_helpers.loop_all_squares():
71              rotation = self.get_rotation_on(bitboard)
72              self._hasher.apply_rotation_hash(bitboard, rotation)
73
74          if self.active_colour == Colour.RED:
75              self._hasher.apply_red_move_hash()
76
77      def flip_colour(self):
78          self.active_colour = self.active_colour.get_flipped_colour()
79
80          if self.active_colour == Colour.RED:
81              self._hasher.apply_red_move_hash()
82
83      def update_move(self, src, dest):
84          piece = self.get_piece_on(src, self.active_colour)
85
```

```
86          self.clear_square(src, Colour.BLUE)
87          self.clear_square(dest, Colour.BLUE)
88          self.clear_square(src, Colour.RED)
89          self.clear_square(dest, Colour.RED)
90
91          self.set_square(dest, piece, self.active_colour)
92
93      def update_rotation(self, src, dest, new_rotation):
94          self.clear_rotation(src)
95          self.set_rotation(dest, new_rotation)
96
97      def clear_rotation(self, bitboard):
98          old_rotation = self.get_rotation_on(bitboard)
99          rotation_1, rotation_2 = self.rotation_bitboards
100         self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.clear_square(
    rotation_1, bitboard)
101         self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.clear_square
    (rotation_2, bitboard)
102
103         self._hasher.apply_rotation_hash(bitboard, old_rotation)
104
105     def clear_square(self, bitboard, colour):
106         piece = self.get_piece_on(bitboard, colour)
107
108         if piece is None:
109             return
110
111         piece_bitboard = self.get_piece_bitboard(piece, colour)
112         colour_bitboard = self.combined_colour_bitboards[colour]
113         all_bitboard = self.combined_all_bitboard
114
115         self.piece_bitboards[colour][piece] = bb_helpers.clear_square(
    piece_bitboard, bitboard)
116         self.combined_colour_bitboards[colour] = bb_helpers.clear_square(
    colour_bitboard, bitboard)
117         self.combined_all_bitboard = bb_helpers.clear_square(all_bitboard,
    bitboard)
118
119         self._hasher.apply_piece_hash(bitboard, piece, colour)
120
121     def set_rotation(self, bitboard, rotation):
122         rotation_1, rotation_2 = self.rotation_bitboards
123         self._hasher.apply_rotation_hash(bitboard, rotation)
124
125         match rotation:
126             case Rotation.UP:
127                 return
128             case Rotation.RIGHT:
129                 self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
    set_square(rotation_1, bitboard)
130                 return
131             case Rotation.DOWN:
132                 self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.
    set_square(rotation_2, bitboard)
133                 return
134             case Rotation.LEFT:
135                 self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
    set_square(rotation_1, bitboard)
136                 self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.
    set_square(rotation_2, bitboard)
137                 return
138             case _:
```

```
139                 raise ValueError('Invalid rotation input (bitboard.py):', rotation
    )
140
141     def set_square(self, bitboard, piece, colour):
142         piece_bitboard = self.get_piece_bitboard(piece, colour)
143         colour_bitboard = self.combined_colour_bitboards[colour]
144         all_bitboard = self.combined_all_bitboard
145
146         self.piece_bitboards[colour][piece] = bb_helpers.set_square(piece_bitboard
    , bitboard)
147         self.combined_colour_bitboards[colour] = bb_helpers.set_square(
    colour_bitboard, bitboard)
148         self.combined_all_bitboard = bb_helpers.set_square(all_bitboard, bitboard)
149
150         self._hasher.apply_piece_hash(bitboard, piece, colour)
151
152     def get_piece_bitboard(self, piece, colour):
153         return self.piece_bitboards[colour][piece]
154
155     def get_piece_on(self, target_bitboard, colour):
156         if not (bb_helpers.is_occupied(self.combined_colour_bitboards[colour],
    target_bitboard)):
157             return None
158
159         return next(
160             (piece for piece in Piece if
161                 bb_helpers.is_occupied(self.get_piece_bitboard(piece, colour),
    target_bitboard)),
162             None)
163
164     def get_rotation_on(self, target_bitboard):
165         rotationBits = [bb_helpers.is_occupied(self.rotation_bitboards[
    RotationIndex.SECONDBIT], target_bitboard), bb_helpers.is_occupied(self.
    rotation_bitboards[RotationIndex.FIRSTBIT], target_bitboard)]
166
167         match rotationBits:
168             case [False, False]:
169                 return Rotation.UP
170             case [False, True]:
171                 return Rotation.RIGHT
172             case [True, False]:
173                 return Rotation.DOWN
174             case [True, True]:
175                 return Rotation.LEFT
176
177     def get_colour_on(self, target_bitboard):
178         for piece in Piece:
179             if self.get_piece_bitboard(piece, Colour.BLUE) & target_bitboard !=
    EMPTY_BB:
180                 return Colour.BLUE
181             elif self.get_piece_bitboard(piece, Colour.RED) & target_bitboard !=
    EMPTY_BB:
182                 return Colour.RED
183
184     def get_piece_count(self, piece, colour):
185         return bb_helpers.pop_count(self.get_piece_bitboard(piece, colour))
186
187     def get_hash(self):
188         return self._hasher.hash
189
190     def convert_to_piece_list(self):
191         piece_list = []
```

```
192
193         for i in range(80):
194             if x := self.get_piece_on(1 << i, Colour.BLUE):
195                 rotation = self.get_rotation_on(1 << i)
196                 piece_list.append((x.upper(), rotation))
197             elif y := self.get_piece_on(1 << i, Colour.RED):
198                 rotation = self.get_rotation_on(1 << i)
199                 piece_list.append((y, rotation))
200             else:
201                 piece_list.append(None)
202
203         return piece_list
```

1.6 CPU

1.6.1 Minimax

minimax.py

```
1  from data.constants import Score, Colour, Miscellaneous
2  from data.states.game.cpu.base import BaseCPU
3  from data.utils.bitboard_helpers import print_bitboard
4  from random import choice
5
6  class MinimaxCPU(BaseCPU):
7      def __init__(self, max_depth, callback, verbose=False):
8          super().__init__(callback, verbose)
9          self._max_depth = max_depth
10
11     def find_move(self, board, stop_event):
12         self.initialise_stats()
13         best_score, best_move = self.search(board, self._max_depth, stop_event)
14
15         if self._verbose:
16             self.print_stats(best_score, best_move)
17
18         self._callback(best_move)
19
20     def search(self, board, depth, stop_event):
21         if (base_case := super().search(board, depth, stop_event)):
22             return base_case
23
24         best_move = None
25
26         if board.get_active_colour() == Colour.BLUE: # is_maximiser
27             max_score = -Score.INFINITE
28
29             for move in board.generate_all_moves(Colour.BLUE):
30                 laser_result = board.apply_move(move)
31
32                 new_score = self.search(board, depth - 1, stop_event)[0]
33
34                 if new_score > max_score:
35                     max_score = new_score
36                     best_move = move
37                 elif new_score == max_score:
38                     choice([best_move, move])
39
40                 board.undo_move(move, laser_result)
41
42             return max_score, best_move
```

```
43
44          else:
45              min_score = Score.INFINITE
46
47              for move in board.generate_all_moves(Colour.RED):
48                  laser_result = board.apply_move(move)
49                  new_score = self.search(board, depth - 1, stop_event)[0]
50
51                  if new_score < min_score:
52                      min_score = new_score
53                      best_move = move
54                  elif new_score == min_score:
55                      choice([best_move, move])
56
57                  board.undo_move(move, laser_result)
58
59              return min_score, best_move
```

1.6.2 Alpha-beta Pruning

alpha_beta.py

```
1  from data.constants import Score, Colour
2  from data.states.game.cpu.base import BaseCPU
3  from random import choice
4
5  class ABMinimaxCPU(BaseCPU):
6      def __init__(self, max_depth, callback, verbose=True):
7          super().__init__(callback, verbose)
8          self._max_depth = max_depth
9
10     def initialise_stats(self):
11         super().initialise_stats()
12         self._stats['beta_prunes'] = 0
13         self._stats['alpha_prunes'] = 0
14
15     def find_move(self, board, stop_event):
16         self.initialise_stats()
17         best_score, best_move = self.search(board, self._max_depth, -Score.
       INFINITE, Score.INFINITE, stop_event)
18
19         if self._verbose:
20             self.print_stats(best_score, best_move)
21
22         self._callback(best_move)
23
24     def search(self, board, depth, alpha, beta, stop_event):
25         if (base_case := super().search(board, depth, stop_event)):
26             return base_case
27
28         best_move = None
29
30         if board.get_active_colour() == Colour.BLUE: # is_maximiser
31             max_score = -Score.INFINITE
32
33             for move in board.generate_all_moves(Colour.BLUE):
34                 laser_result = board.apply_move(move)
35                 new_score = self.search(board, depth - 1, alpha, beta, stop_event)
       [0]
36
37                 if new_score > max_score:
```

```
38                      max_score = new_score
39                      best_move = move
40
41                  board.undo_move(move, laser_result)
42
43                  alpha = max(alpha, max_score)
44
45                  if beta <= alpha:
46                      self._stats['alpha_prunes'] += 1
47                      break
48
49              return max_score, best_move
50
51          else:
52              min_score = Score.INFINITE
53
54              for move in board.generate_all_moves(Colour.RED):
55                  laser_result = board.apply_move(move)
56                  new_score = self.search(board, depth - 1, alpha, beta, stop_event)
    [0]
57
58                  if new_score < min_score:
59                      min_score = new_score
60                      best_move = move
61
62                  board.undo_move(move, laser_result)
63
64                  beta = min(beta, min_score)
65                  if beta <= alpha:
66                      self._stats['beta_prunes'] += 1
67                      break
68
69              return min_score, best_move
70
71  class ABNegamaxCPU(BaseCPU):
72      def __init__(self, max_depth, callback, verbose=True):
73          super().__init__(callback, verbose)
74          self._max_depth = max_depth
75
76      def initialise_stats(self):
77          super().initialise_stats()
78          self._stats['beta_prunes'] = 0
79
80      def find_move(self, board, stop_event):
81          self.initialise_stats()
82          best_score, best_move = self.search(board, self._max_depth, -Score.
    INFINITE, Score.INFINITE, stop_event)
83
84          if self._verbose:
85              self.print_stats(best_score, best_move)
86
87          self._callback(best_move)
88
89      def search(self, board, depth, alpha, beta, stop_event):
90          if (base_case := super().search(board, depth, stop_event, absolute=True)):
91              return base_case
92
93          best_move = None
94          best_score = alpha
95
96          for move in board.generate_all_moves(board.get_active_colour()):
97              laser_result = board.apply_move(move)
```

```
98
99              new_score = self.search(board, depth - 1, -beta, -best_score,
    stop_event)[0]
100             new_score = -new_score
101
102             if new_score > best_score:
103                 best_score = new_score
104                 best_move = move
105             elif new_score == best_score:
106                 best_move = choice([best_move, move])
107
108             board.undo_move(move, laser_result)
109
110             if best_score >= beta:
111                 self._stats['beta_prunes'] += 1
112                 break
113
114         return best_score, best_move
```

1.6.3 Transposition Table CPU

alpha_beta.py

```
1  from data.states.game.cpu.transposition_table import TranspositionTable
2  from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
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         hash = board.to_hash()
11         score, move = self._table.get_entry(hash, depth, alpha, beta)
12
13         if score is not None:
14             self._stats['cache_hits'] += 1
15             self._stats['nodes'] += 1
16
17             return score, move
18         else:
19             score, move = super().search(board, depth, alpha, beta, stop_event)
20             self._table.insert_entry(score, move, hash, depth, alpha, beta)
21
22             return score, move
23
24 class TTMinimaxCPU(TranspositionTableMixin, ABMinimaxCPU):
25     def initialise_stats(self):
26         super().initialise_stats()
27         self._stats['cache_hits'] = 0
28
29     def print_stats(self, score, move):
30         self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] /
    self._stats['nodes'], 3)
31         self._stats['cache_entries'] = len(self._table._table)
32         super().print_stats(score, move)
33
34 class TTNegamaxCPU(TranspositionTableMixin, ABNegamaxCPU):
35     def initialise_stats(self):
36         super().initialise_stats()
37         self._stats['cache_hits'] = 0
```

```
38
39      def print_stats(self, score, move):
40          self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] /
    self._stats['nodes'], 3)
41          self._stats['cache_entries'] = len(self._table._table)
42          super().print_stats(score, move)
```

1.6.4 Evaluator

evaluator.py

```
1  from data.constants import Colour, Piece, Score
2  from data.utils.bitboard_helpers import index_to_bitboard, pop_count,
       occupied_squares, bitboard_to_index
3  from data.states.game.components.psqt import PSQT, FLIP
4  import random
5  from data.managers.logs import initialise_logger
6
7  logger = initialise_logger(__name__)
8
9  class Evaluator:
10     def __init__(self, verbose=True):
11         self._verbose = verbose
12         pass
13
14     def evaluate(self, board, absolute=False):
15         #Add tapered evaluation
16         blue_score = self.evaluate_pieces(board, Colour.BLUE) + self.
       evaluate_position(board, Colour.BLUE) + self.evaluate_mobility(board, Colour.
       BLUE) + self.evaluate_pharoah_safety(board, Colour.BLUE)
17
18         red_score = self.evaluate_pieces(board, Colour.RED) + self.
       evaluate_position(board, Colour.RED) + self.evaluate_mobility(board, Colour.
       RED) + self.evaluate_pharoah_safety(board, Colour.RED)
19
20         if (self._verbose):
21             logger.info('\nPosition:', self.evaluate_position(board, Colour.BLUE),
        self.evaluate_position(board, Colour.RED))
22             logger.info('Mobility:', self.evaluate_mobility(board, Colour.BLUE),
       self.evaluate_mobility(board, Colour.RED))
23             logger.info('Safety:', self.evaluate_pharoah_safety(board, Colour.BLUE
       ), self.evaluate_pharoah_safety(board, Colour.RED))
24             logger.info('Overall score', blue_score - red_score)
25
26         if absolute and board.get_active_colour() == Colour.RED:
27             return red_score - blue_score
28
29         return blue_score - red_score
30
31     def evaluate_pieces(self, board, colour):
32         # return random.randint(-100, 100)
33         return (
34             Score.SPHINX * board.bitboards.get_piece_count(Piece.SPHINX, colour) +
35             Score.PYRAMID * board.bitboards.get_piece_count(Piece.PYRAMID, colour)
        +
36             Score.ANUBIS * board.bitboards.get_piece_count(Piece.ANUBIS, colour) +
37             Score.SCARAB * board.bitboards.get_piece_count(Piece.SCARAB, colour)
38         )
39
40     def evaluate_position(self, board, colour):
41         score = 0
```

```
42
43          for piece in Piece:
44              if piece == Piece.SPHINX:
45                  continue
46
47              for colour in Colour:
48                  piece_bitboard = board.bitboards.get_piece_bitboard(piece, colour)
49
50                  for bitboard in occupied_squares(piece_bitboard):
51                      index = bitboard_to_index(bitboard)
52                      index = FLIP[index] if colour == Colour.BLUE else index
53
54                      score += PSQT[piece][index] * Score.POSITION
55
56          return score
57
58      def evaluate_mobility(self, board, colour):
59          number_of_moves = pop_count(board.get_all_valid_squares(colour))
60
61          return number_of_moves * Score.MOVE
62
63      def evaluate_pharoah_safety(self, board, colour):
64          pharoah_bitboard = board.bitboards.get_piece_bitboard(Piece.PHAROAH,
    colour)
65          pharoah_available_moves = pop_count(board.get_valid_squares(
    pharoah_bitboard, colour))
66          return (8 - pharoah_available_moves) * Score.PHAROAH_SAFETY
```

1.6.5 Multithreading

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         self.stop_cpu()
20         self._running = False
21
22     def stop_cpu(self):
23         self._stop_event.set()
24         self._board = None
25
26     def start_cpu(self, board):
27         self._stop_event.clear()
28         self._board = board
29
```

```
    def run(self):
        while self._running:
            if self._board and self._cpu:
                self._cpu.find_move(self._board, self._stop_event)
                self.stop_cpu()
            else:
                time.sleep(1)
                if self._verbose:
                    logger.debug(f'(CPUThread.run) Thread {threading.get_native_id
    ()} idling...')
```

1.6.6 Zobrist Hashing

zobrist_hasher.py

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

zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)] # 10
     pieces + 4 rotations, 8 y, 10
red_move_hash = randint(0, 2 ** 64)

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

rotation_lookup = {
    rotation: i + 10 for i, rotation in enumerate(Rotation)
}

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

    def get_piece_hash(self, index, piece, colour):
        piece_index = piece_lookup[colour][piece]
        return zobrist_table[index][piece_index]

    def get_rotation_hash(self, index, rotation):
        rotation_index = rotation_lookup[rotation]
        return zobrist_table[index][rotation_index]

    def apply_piece_hash(self, bitboard, piece, colour):
        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):
        index = bitboard_to_index(bitboard)
        rotation_hash = self.get_rotation_hash(index, rotation)
        self.hash ^= rotation_hash

    def apply_red_move_hash(self):
        self.hash ^= red_move_hash
```

1.6.7 Transposition Table

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):
        # return hash_key % self._max_entries
        return str(hash_key)

    def insert_entry(self, score, move, hash_key, depth, alpha, beta):
        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 FIRST ADDED ENTRY 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):
        index = self.calculate_entry_index(hash_key)

        if index not in self._table:
            return None, None

        entry = self._table[index]

        if entry.hash_key == hash_key and entry.depth >= depth:
            if entry.flag == TranspositionFlag.EXACT:
                return entry.score, entry.move

            if entry.flag == TranspositionFlag.LOWER and entry.score >= beta:
                return entry.score, entry.move

            if entry.flag == TranspositionFlag.UPPER and entry.score <= alpha:
                return entry.score, entry.move

```

```
57          return None, None
```

1.7 Database

1.7.1 DDL

create_games_table_19112024.py

```
import sqlite3
from pathlib import Path

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

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

    cursor.execute('''
        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():
    connection = sqlite3.connect(database_path)
    cursor = connection.cursor()

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

    connection.commit()
    connection.close()

upgrade()
# downgrade()
```

change_fen_string_column_name_23122024.py

```
import sqlite3
from pathlib import Path

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

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

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

```
    ''')

    connection.commit()
    connection.close()

def downgrade():
    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()
```

1.7.2 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):
    connection = sqlite3.connect(database_path, detect_types=sqlite3.
    PARSE_DECLTYPES)
    cursor = connection.cursor()

    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():
    connection = sqlite3.connect(database_path, detect_types=sqlite3.
    PARSE_DECLTYPES)
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()

    cursor.execute('''
        SELECT * FROM games
    ''')
    games = cursor.fetchall()

    connection.close()

    return [dict(game) for game in games]

def delete_all_games():
```

```
36      connection = sqlite3.connect(database_path)
37      cursor = connection.cursor()
38
39      cursor.execute('''
40          DELETE FROM games
41      ''')
42
43      connection.commit()
44      connection.close()
45
46  def delete_game(id):
47      connection = sqlite3.connect(database_path)
48      cursor = connection.cursor()
49
50      cursor.execute('''
51          DELETE FROM games WHERE id = ?
52      ''', (id,))
53
54      connection.commit()
55      connection.close()
56
57  def get_ordered_games(column, ascend=True, start_row=1, end_row=10):
58      if not isinstance(ascend, bool) or not isinstance(column, str):
59          raise ValueError('(database_helpers.get_ordered_games) Invalid input
        arguments!')
60
61      connection = sqlite3.connect(database_path, detect_types=sqlite3.
        PARSE_DECLTYPES)
62      connection.row_factory = sqlite3.Row
63      cursor = connection.cursor()
64
65      if ascend:
66          ascend_arg = 'ASC'
67      else:
68          ascend_arg = 'DESC'
69
70      if column == 'winner':
71          cursor.execute(f'''
72              SELECT * FROM
73                  (SELECT ROW_NUMBER() OVER (
74                      PARTITION BY winner
75                      ORDER BY time {ascend_arg}, number_of_ply {ascend_arg}
76                  ) AS row_num, * FROM games)
77              WHERE row_num >= ? AND row_num <= ?
78          ''', (start_row, end_row))
79      else:
80          cursor.execute(f'''
81              SELECT * FROM
82                  (SELECT ROW_NUMBER() OVER (
83                      ORDER BY {column} {ascend_arg}
84                  ) AS row_num, * FROM games)
85              WHERE row_num >= ? AND row_num <= ?
86          ''', (start_row, end_row))
87
88      games = cursor.fetchall()
89
90      connection.close()
91
92      return [dict(game) for game in games]
93
94  def get_number_of_games():
95      connection = sqlite3.connect(database_path)
```

```
    cursor = connection.cursor()

    cursor.execute("""
        SELECT COUNT(ROWID) FROM games
    """)

    result = cursor.fetchall()[0][0]

    connection.close()

    return result

# delete_all_games()
```

1.8 Shaders

1.8.1 Shader Manager

Uses interface protocol! `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)
```

```
42          self._shader_stack = [ShaderType.BASE]
43
44          self._vert_shaders = {}
45          self._frag_shaders = {}
46          self._programs = {}
47          self._vaos = {}
48          self._textures = {}
49          self._shader_passes = {}
50          self.framebuffers = {}
51
52          self.load_shader(ShaderType.BASE)
53          self.load_shader(ShaderType._CALIBRATE)
54          self.create_framebuffer(ShaderType._CALIBRATE)
55
56      def load_shader(self, shader_type, **kwargs):
57          self._shader_passes[shader_type] = shader_pass_lookup[shader_type](self,
    **kwargs)
58
59          self.create_vao(shader_type)
60
61      def clear_shaders(self):
62          self._shader_stack = [ShaderType.BASE]
63
64      def create_vao(self, shader_type):
65          frag_name = shader_type[1:] if shader_type[0] == '_' else shader_type
66          vert_path = Path(shader_path / 'vertex/base.vert').resolve()
67          frag_path = Path(shader_path / f'fragments/{frag_name}.frag').resolve()
68
69          self._vert_shaders[shader_type] = vert_path.read_text()
70          self._frag_shaders[shader_type] = frag_path.read_text()
71
72          program = self._ctx.program(vertex_shader=self._vert_shaders[shader_type],
     fragment_shader=self._frag_shaders[shader_type])
73          self._programs[shader_type] = program
74
75          if shader_type == ShaderType._CALIBRATE:
76              self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._pygame_buffer, '2f 2f', 'vert', 'texCoords')])
77          else:
78              self._vaos[shader_type] = self._ctx.vertex_array(self._programs[
    shader_type], [(self._opengl_buffer, '2f 2f', 'vert', 'texCoords')])
79
80      def create_framebuffer(self, shader_type, size=None, filter=moderngl.NEAREST):
81          texture_size = size or self._screen_size
82          texture = self._ctx.texture(size=texture_size, components=4)
83          texture.filter = (filter, filter)
84
85          self._textures[shader_type] = texture
86          self.framebuffers[shader_type] = self._ctx.framebuffer(color_attachments=[
    self._textures[shader_type]])
87
88      def render_to_fbo(self, shader_type, texture, output_fbo=None, program_type=
    None, use_image=True, **kwargs):
89          fbo = output_fbo or self.framebuffers[shader_type]
90          program = self._programs[program_type] if program_type else self._programs
    [shader_type]
91          vao= self._vaos[program_type] if program_type else self._vaos[shader_type]
92
93          fbo.use()
94          texture.use(0)
95
96          if use_image:
```

```
97              program['image'] = 0
98          for uniform, value in kwargs.items():
99              program[uniform] = value
100
101         vao.render(mode=moderngl.TRIANGLE_STRIP)
102
103     def apply_shader(self, shader_type, **kwargs):
104         if shader_type in self._shader_stack:
105             return
106             raise ValueError('(ShaderManager) Shader already being applied!',
    shader_type)
107
108         self.load_shader(shader_type, **kwargs)
109         self._shader_stack.append(shader_type)
110
111         self._shader_stack.sort(key=lambda shader: -SHADER_PRIORITY.index(shader))
112
113     def remove_shader(self, shader_type):
114         if shader_type in self._shader_stack:
115             self._shader_stack.remove(shader_type)
116
117     def render_output(self, texture):
118         output_shader_type = self._shader_stack[-1]
119         self._ctx.screen.use() # IMPORTANT
120
121         self.get_fbo_texture(output_shader_type).use(0)
122         self._programs[output_shader_type]['image'] = 0
123
124         self._vaos[output_shader_type].render(mode=moderngl.TRIANGLE_STRIP) #
    SOMETHING ABOUT DRAWING FLIPS THE
125
126     def get_fbo_texture(self, shader_type):
127         return self.framebuffers[shader_type].color_attachments[0]
128
129     def calibrate_pygame_surface(self, pygame_surface):
130         texture = self._ctx.texture(pygame_surface.size, 4)
131         texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
132         texture.swizzle = 'BGRA'
133         texture.write(pygame_surface.get_view('1'))
134
135         self.render_to_fbo(ShaderType._CALIBRATE, texture)
136
137         return self.get_fbo_texture(ShaderType._CALIBRATE)
138
139     def draw(self, surface, arguments):
140         self._ctx.viewport = (0, 0, *self._screen_size)
141         texture = self.calibrate_pygame_surface(surface)
142
143         for shader_type in self._shader_stack:
144             self._shader_passes[shader_type].apply(texture, **arguments.get(
    shader_type, {}))
145             texture = self.get_fbo_texture(shader_type)
146
147         self.render_output(texture)
148
149     def __del__(self):
150         self.cleanup()
151
152     def cleanup(self):
153         self._pygame_buffer.release()
154         self._opengl_buffer.release()
155         for program in self._programs:
```

```
156             self._programs[program].release()
157         for texture in self._textures:
158             self._textures[texture].release()
159         for vao in self._vaos:
160             self._vaos[vao].release()
161         for framebuffer in self.framebuffers:
162             self.framebuffers[framebuffer].release()
163 
164     def handle_resize(self, new_screen_size):
165         self._screen_size = new_screen_size
166 
167         for shader_type in self.framebuffers:
168             filter = self._textures[shader_type].filter[0]
169             self.create_framebuffer(shader_type, size=self._screen_size, filter=
    filter) # RECREATE FRAMEBUFFER TO PREVENT SCALING ISSUES
```

1.8.2 Rays

occlusion.frag

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

shadowmap.frag

```
1  # version 330 core
2  
3  in vec2 uvs;
4  out vec4 f_colour;
5  
6  uniform sampler2D image;
7  uniform float resolution;
8  
9  #define PI 3.1415926536;
10 const float THRESHOLD = 0.99;
11 
12 // void main() {
13 //     f_colour = vec4(texture(image, uvs).rgba);
14 // }
15 
16 // float get_colour(float angle, float radius) {
17 //     for (float currentRadius=0 ; currentRadius < radius ; currentRadius += 
    0.01) {
18 //         vec2 coords = vec2(-currentRadius * sin(angle), -currentRadius * cos(
    angle)) / 2.0 + 0.5;
```

```
//          vec4 colour = texture(image, coords);

//          if (colour.r == 1.0) {
//              // return 1.0;
//              return 0.9;
//          }
//      }

//      return 0.5;
// }

// void main() {
//      float distance = 1.0;

//          // rectangular to polar filter
//      vec2 norm = uvs.xy * 2.0 - 1.0; // [0, 1] -> [-1, 1]
//      float angle = atan(norm.y, norm.x); // range [pi, -pi]       [1, 0] = 0, [-1, 0] = pi or -pi
//      float radius = length(norm);

//      // 0.5, 1 -> 0, 0.5
//      // 1, 0.5 -> 0.5, 0


//      // coord which we will sample from occlude map
//      vec2 polar_coords = vec2(-radius * sin(angle), -radius * cos(angle)) / 2.0 + 0.5; // .s == .x, .t == .y

//      // for (float y = 0.0; y < resolution.y; y++) {
//          //sample the occlusion map
//          // float norm_distance = y / resolution.y;
//          // vec4 data = texture(image, polar_coords).rgba;

//          //the current distance is how far from the top we've come

//          //if we've hit an opaque fragment (occluder), then get new distance
//          //if the new distance is below the current, then we'll use that for our ray

//          // if (data.a == 1.0) {
//          //      distance = min(distance, polar_coords.y);
//              // distance = norm_distance;
//              // break;
//          // } // if using return, does not set frag colour so just returns normal image
//      // }

//      // float brightness = get_colour(angle, radius);
//      // f_colour = vec4(vec3(brightness), 1.0);

//      f_colour = texture(image, polar_coords).rgba;
// }


// void main() {
//   float distance = 0.5;
//   float resolution = 256;

//      for (float y=0.0; y< resolution; y+=1.0) { // putting y < resolution.y doesn't work for some reason
//          //rectangular to polar filter
//          vec2 norm = vec2(uvs.s, y/resolution) * 2.0 - 1.0;
```

```
//          float theta = PI*1.5 + norm.x * PI;
//          float r = (1.0 + norm.y) * 0.5;

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

//          //sample the occlusion map
//          vec4 data = texture(image, coord);

//          //the current distance is how far from the top we've come
//          float dst = y/resolution;

//          //if we've hit an opaque fragment (occluder), then get new distance
//          //if the new distance is below the current, then we'll use that for our ray
//          float caster = data.r;
//          if (caster > THRESHOLD) {
//              distance = 1.0;
//              // distance = min(distance, dst);
//              break;
//              //NOTE: we could probably use "break" or "return" here
//          }
//          distance = min(distance, dst);
//      }

//      f_colour = vec4(vec3(distance), 1.0);
// }


void main() {
  float distance = 1.0;

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

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

        // float radius = length(norm);

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

        //sample the occlusion map
        vec4 data = texture(image, coords);

        //the current distance is how far from the top we've come

        //if we've hit an opaque fragment (occluder), then get new distance
        //if the new distance is below the current, then we'll use that for our ray
        // float caster = data.r;
        // if (caster >= THRESHOLD) {
        //      distance = min(distance, dst);
        //      break;
        // }
        distance = max(distance * step(data.r, THRESHOLD), min(distance, dst));
    }

```

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



// void main() {
//     vec2 norm = vec2(uvs.x, uvs.y) * 2.0 - 1.0;
//     float angle = (1.5 + norm.x) * PI;
//     float radius = (1.0 + norm.y) * 0.5;
//     vec2 coords = vec2(-radius * sin(angle), -radius * cos(angle)) / 2.0 + 0.5;

//     vec4 data = texture(image, coords);

//     f_colour = vec4(data.rgb, 1.0);
// }
```

lightmap.frag

```
# version 330 core

#define PI 3.14159265

//inputs from vertex shader
in vec2 uvs;
out vec4 f_colour;

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

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

//sample from the 1D distance map
float sample(vec2 coord, float r) {
  return step(r, texture(image, coord).r); // returns 1.0 if 2nd parameter greater
     than 1st, 0.0 if not
}

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

  //the tex coord to sample our 1D lookup texture
  //always 0.0 on y axis
  vec2 tc = vec2(coord, 0.0);

  //the center tex coord, which gives us hard shadows
  float center = sample(tc, r); // center = 1.0 -> in light, center = 0.0, -> in
    shadow
  center = center * step(angle, radiansClamp.y) * step(radiansClamp.x, angle);

  //we multiply the blur amount by our distance from center
```

```
  //this leads to more blurriness as the shadow "fades away"
    // straight to cuved edges
  float blur = (1.0 / resolution) * smoothstep(0.0, 0.1, r);

  //now we use a simple gaussian blur
  float sum = 0.0;

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

  sum += center * 0.16;

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

  //sum of 1.0 -> in light, 0.0 -> in shadow

  //multiply the summed amount by our distance, which gives us a radial falloff
  // //then multiply by vertex (light) color
    // if (center == 1.0) {
  float isLit = mix(center, sum, softShadow);

  // vec3 final_colour = vec3(texture(image, uvs).rgb * vec3(sum * smoothstep(1.0,
     0.0, r)) * 5);

  // f_colour = vec4(final_colour.r + texture(occlusionMap, uvs).r, final_colour.
    gb, 1.0);
  f_colour = vec4(normLightColour, isLit * smoothstep(1.0, falloff, r));
    // } else {
    //     f_colour = vec4(0.0, 1.0, 0.0, 1.0);
    // }
}

// void main() {
//     f_colour = vec4(texture(image, uvs).rgb, 1.0);
// }
```

1.8.3 Bloom

highlight_colour.frag

```
# version 330 core

uniform sampler2D image;
uniform sampler2D highlight;

uniform vec3 colour;
uniform float threshold;
uniform float intensity;

in vec2 uvs;
out vec4 f_colour;

vec3 normColour = colour / 255;

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

```
    float isClose = step(abs(pixel.r - normColour.r), threshold) * step(abs(pixel.g - normColour.g), threshold) * step(abs(pixel.b - normColour.b), threshold);

    if (isClose == 1.0) {
        f_colour = vec4(vec3(pixel.rgb * intensity), 1.0);
    } else {
        f_colour = vec4(texture(highlight, uvs).rgb, 1.0);
    }
}
```

blur.frag

```
#version 330 core

uniform sampler2D image;

in vec2 uvs;
out vec4 f_colour;

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);
}
```

blur.frag

```
#version 330 core

uniform sampler2D image;

in vec2 uvs;
out vec4 f_colour;

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

```

```
12 void main()
13 {
14     vec2 offset = 1.0 / textureSize(image, 0);
15     vec3 result = texture(image, uvs).rgb * weight[0];
16 
17     if (horizontal) {
18         for (int i = 1 ; i < passes ; ++i) {
19             result += texture(image, uvs + vec2(offset.x * i, 0.0)).rgb * weight[i
       ];
20             result += texture(image, uvs - vec2(offset.x * i, 0.0)).rgb * weight[i
       ];
21         }
22     }
23     else {
24         for (int i = 1 ; i < passes ; ++i) {
25             result += texture(image, uvs + vec2(0.0, offset.y * i)).rgb * weight[i
       ];
26             result += texture(image, uvs - vec2(0.0, offset.y * i)).rgb * weight[i
       ];
27         }
28     }
29     f_colour = vec4(result, 1.0);
30 }
```