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.

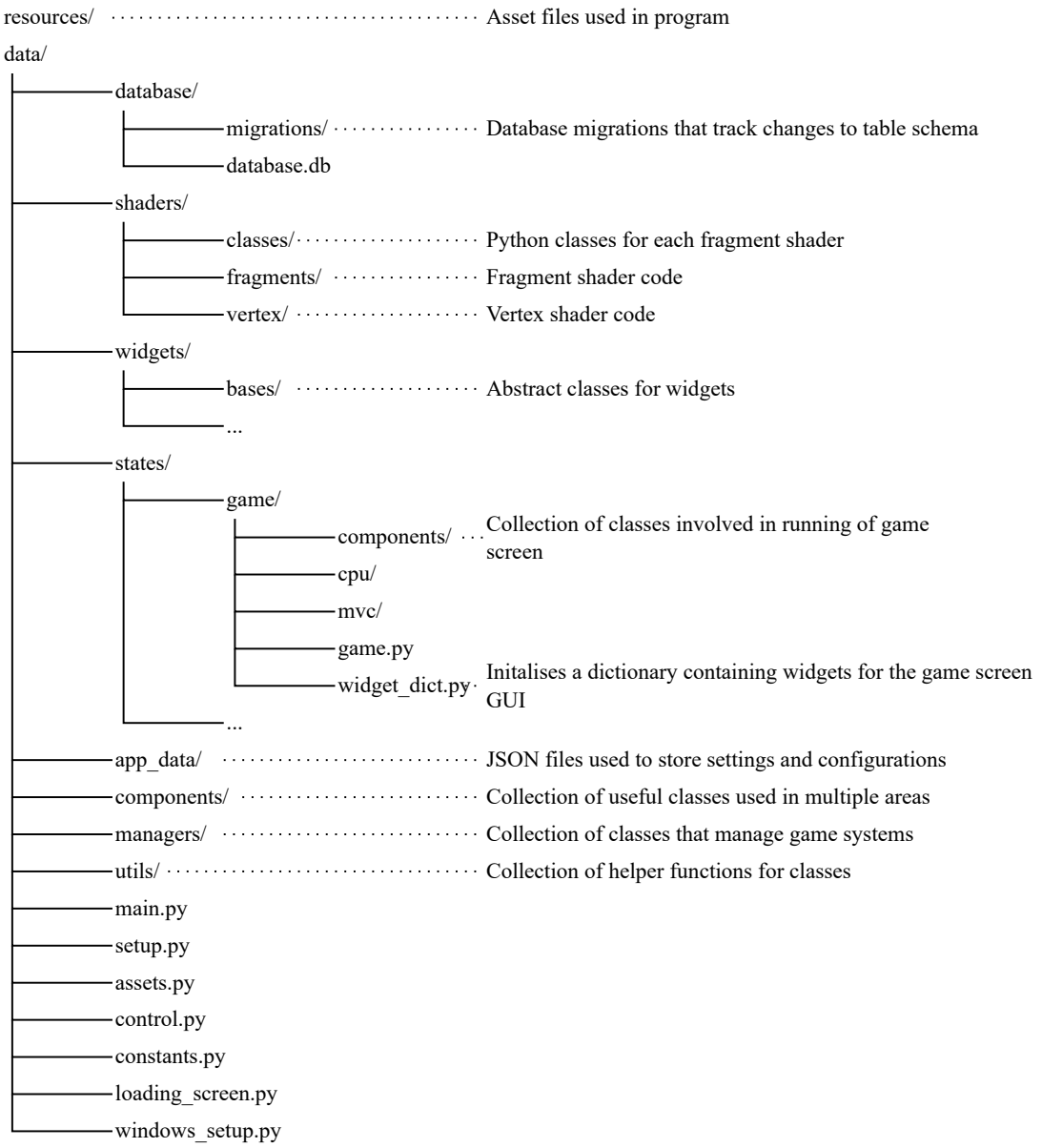

Figure 1: File tree diagram

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

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

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

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

        self.run()

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

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

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

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

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

    def draw(self):
```

```
        """
        Draws logo to screen.
        """
        window.screen.fill((0, 0, 0))

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

        window.update()

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

1.3.3 Helper functions

These files provide useful functions for different classes.
`asset_helpers.py (Functions used for assets and pygame Surfaces)`

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

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

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

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

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

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

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

        first_frame = image.copy().convert('RGBA')
        yield first_frame
        image.seek(1)

        while True:
            current_frame = image.copy()
            yield current_frame
            image.seek(image.tell() + 1)
    except EOFError:
        pass

def get_perimeter_sample(image_size, number):
    """
    Used for particle drawing class, generates roughly equally distributed points around a rectangular image surface's perimeter.

    Args:
        image_size (tuple[float, float]): Image surface size.
        number (int): Number of points to be generated.

    Returns:
        list[tuple[int, int], ...]: List of random points on perimeter of image surface.
    """
    perimeter = 2 * (image_size[0] + image_size[1])
    # Flatten perimeter to a single number representing the distance from the top-middle of the surface going clockwise, and create a list of equally spaced points
    perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in range(0, number)]
    pos_list = []

    for perimeter_offset in perimeter_offsets:
        # For every point, add a random offset
        max_displacement = int(perimeter / (number * 4))
        perimeter_offset += randint(-max_displacement, max_displacement)

        if perimeter_offset > perimeter:
            perimeter_offset -= perimeter

        # Convert 1D distance back into 2D points on image surface perimeter
        if perimeter_offset < image_size[0]:
            pos_list.append((perimeter_offset, 0))
        elif perimeter_offset < image_size[0] + image_size[1]:
            pos_list.append((image_size[0], perimeter_offset - image_size[0]))
        elif perimeter_offset < image_size[0] + image_size[1] + image_size[0]:
            pos_list.append((perimeter_offset - image_size[0] - image_size[1], image_size[1]))
        else:
            pos_list.append((0, perimeter - perimeter_offset))
    return pos_list
```

```
93
94  def get_angle_between_vectors(u, v, deg=True):
95      """
96      Uses the dot product formula to find the angle between two vectors.
97
98      Args:
99          u (list[int, int]): Vector 1.
100         v (list[int, int]): Vector 2.
101         deg (bool, optional): Return results in degrees. Defaults to True.
102
103     Returns:
104         float: Angle between vectors.
105     """
106     dot_product = sum(i * j for (i, j) in zip(u, v))
107     u_magnitude = math.sqrt(u[0] ** 2 + u[1] ** 2)
108     v_magnitude = math.sqrt(v[0] ** 2 + v[1] ** 2)
109
110     cos_angle = dot_product / (u_magnitude * v_magnitude)
111     radians = math.acos(min(max(cos_angle, -1), 1))
112
113     if deg:
114         return math.degrees(radians)
115     else:
116         return radians
117
118 def get_rotational_angle(u, v, deg=True):
119     """
120     Get bearing angle relative to positive x-axis centered on second vector.
121
122     Args:
123         u (list[int, int]): Vector 1.
124         v (list[int, int]): Vector 2, set as center of axes.
125         deg (bool, optional): Return results in degrees. Defaults to True.
126
127     Returns:
128         float: Bearing angle between vectors.
129     """
130     radians = math.atan2(u[1] - v[1], u[0] -v[0])
131
132     if deg:
133         return math.degrees(radians)
134     else:
135         return radians
136
137 def get_vector(src_vertex, dest_vertex):
138     """
139     Get vector describing translation between two points.
140
141     Args:
142         src_vertex (list[int, int]): Source vertex.
143         dest_vertex (list[int, int]): Destination vertex.
144
145     Returns:
146         tuple[int, int]: Vector between the two points.
147     """
148     return (dest_vertex[0] - src_vertex[0], dest_vertex[1] - src_vertex[1])
149
150 def get_next_corner(vertex, image_size):
151     """
152     Used in particle drawing system, finds coordinates of the next corner going
      clockwise, given a point on the perimeter.
153
```

```
    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):
```

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

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

```
1  import json
2  from pathlib import Path
3  
4  module_path = Path(__file__).parent
5  default_file_path = (module_path / '../app_data/default_settings.json').resolve()
6  user_file_path = (module_path / '../app_data/user_settings.json').resolve()
7  themes_file_path = (module_path / '../app_data/themes.json').resolve()
8  
9  def load_json(path):
10     """
11     Args:
12         path (str): Path to JSON file.
13 
14     Raises:
15         Exception: Invalid file.
16 
17     Returns:
18         dict: Parsed JSON file.
19     """
20     try:
21         with open(path, 'r') as f:
22             file = json.load(f)
23 
24         return file
25     except:
26         raise Exception('Invalid JSON file (data_helpers.py)')
27 
28 def get_user_settings():
29     return load_json(user_file_path)
30 
31 def get_default_settings():
32     return load_json(default_file_path)
```

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

widget_helpers.py (Files used for creating widgets)

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

```
        pygame.Surface: Slider image surface.
    """
    gradient_surface = pygame.Surface(size, pygame.SRCALPHA)

    first_round_end = gradient_surface.height / 2
    second_round_end = gradient_surface.width - first_round_end
    gradient_y_mid = gradient_surface.height / 2

    # Iterate through length of slider
    for i in range(gradient_surface.width):
        draw_height = gradient_surface.height

        if i < first_round_end or i > second_round_end:
            # Draw semicircular caps if x-distance less than or greater than radius of cap (half of image height)
            distance_from_cutoff = min(abs(first_round_end - i), abs(i - second_round_end))
            draw_height = calculate_gradient_slice_height(distance_from_cutoff, gradient_surface.height / 2)

        # Get colour from distance from left side of slider
        color = pygame.Color(0)
        color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)

        draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
        draw_rect.center = (i, gradient_y_mid)

        pygame.draw.rect(gradient_surface, color, draw_rect)

    border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.height))
    pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(border_width), border_radius=int(size[1] / 2))

    return gradient_surface

def calculate_gradient_slice_height(distance, radius):
    """
    Calculate height of vertical slice of semicircular slider cap.

    Args:
        distance (float): x-distance from center of circle.
        radius (float): Radius of semicircle.

    Returns:
        float: Height of vertical slice.
    """
    return sqrt(radius ** 2 - distance ** 2) * 2 + 2

def create_slider_thumb(radius, colour, border_colour, border_width):
    """
    Creates surface with bordered circle.

    Args:
        radius (float): Radius of circle.
        colour (pygame.Color): Fill colour.
        border_colour (pygame.Color): Border colour.
        border_width (float): Border width.

    Returns:
        pygame.Surface: Circle surface.
    """
```

```
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     """
```

```
    Creates bordered textbox with shadow, flat, and highlighted vertical regions.

    Args:
        size (list[int, int]): Image size.
        border_width (float): Border width.
        colours (list[pygame.Color, ...]): List of 4 colours, representing border colour, shadow colour, flat colour and highlighted colour.

    Returns:
        pygame.Surface: Textbox surface.
    """
    surface = pygame.Surface(size, pygame.SRCALPHA)

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

    return surface
```

1.3.4 Theme

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

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.

        Raises:
            KeyError: Instance does not have requested attribute.

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

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

        return item

theme = ThemeManager()
```

1.4 GUI

1.4.1 Laser

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

laser_draw.py

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

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

GLOW_SCALE_FACTOR = 1.5

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

    @property
    def firing(self):
        return len(self._laser_lists) > 0

```

```
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.
```

```
            board_size (tuple[int, int]): The new size of the board.
        """
        self._board_position = board_position
        self._square_size = board_size[0] / 10
```

1.4.2 Particles

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

particles_draw.py

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

```
45              cropped_image.fill((0, 0, 0, 0))
46              cropped_image.blit(image, (0, 0))
47
48              corners_to_draw = None
49
50              if vertex_1[0] == vertex_2[0] or vertex_1[1] == vertex_2[1]: # Points
    on the same side
51                  corners_to_draw = 4
52
53              elif abs(vertex_1[0] - vertex_2[0]) == image_size[0] or abs(vertex_1
    [1] - vertex_2[1]) == image_size[1]: # Points on opposite sides
54                  corners_to_draw = 2
55
56              elif angle < 180: # Points on adjacent sides
57                  corners_to_draw = 3
58
59              else:
60                  corners_to_draw = 1
61
62              corners_list = []
63              for j in range(corners_to_draw):
64                  if len(corners_list) == 0:
65                      corners_list.append(get_next_corner(vertex_2, image_size))
66                  else:
67                      corners_list.append(get_next_corner(corners_list[-1],
    image_size))
68
69              pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *
    corners_list, vertex_1))
70
71              fragment_list.append(cropped_image)
72
73          return fragment_list
74
75      def add_captured_piece(self, piece, colour, rotation, position, size):
76          """
77          Adds a captured piece to fragment into particles.
78
79          Args:
80              piece (Piece): The piece type.
81              colour (Colour.BLUE | Colour.RED): The active colour of the piece.
82              rotation (int): The rotation of the piece.
83              position (tuple[int, int]): The position where particles originate
    from.
84              size (tuple[int, int]): The size of the piece.
85          """
86          piece_sprite = PieceSprite(piece, colour, rotation)
87          piece_sprite.set_geometry((0, 0), size)
88          piece_sprite.set_image()
89
90          particles = self.fragment_image(piece_sprite.image, 5)
91
92          for particle in particles:
93              self.add_particle(particle, position)
94
95      def add_sparks(self, radius, colour, position):
96          """
97          Adds laser spark particles.
98
99          Args:
100             radius (int): The radius of the sparks.
101             colour (Colour.BLUE | Colour.RED): The active colour of the sparks.
```

```
102             position (tuple[int, int]): The position where particles originate
    from.
103         """
104         for i in range(randint(10, 15)):
105             velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
106             random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in
    colour]
107             self._particles.append([None, [radius, random_colour], [*position],
    velocity, 0])
108 
109     def add_particle(self, image, position):
110         """
111         Adds a particle.
112 
113         Args:
114             image (pygame.Surface): The image of the particle.
115             position (tuple): The position of the particle.
116         """
117         velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
118 
119         # Each particle is stored with its attributes: [surface, copy of surface,
    position, velocity, lifespan]
120         self._particles.append([image, image.copy(), [*position], velocity, 0])
121 
122     def update(self):
123         """
124         Updates each particle and its attributes.
125         """
126         for i in range(len(self._particles) - 1, -1, -1):
127             particle = self._particles[i]
128 
129             #update position
130             particle[2][0] += particle[3][0]
131             particle[2][1] += particle[3][1]
132 
133             #update lifespan
134             self._particles[i][4] += 0.01
135 
136             if self._particles[i][4] >= 1:
137                 self._particles.pop(i)
138                 continue
139 
140             if isinstance(particle[1], pygame.Surface): # Particle is a piece
141                 # Update velocity
142                 particle[3][1] += self._gravity
143 
144                 # Update size
145                 image_size = particle[1].get_rect().size
146                 end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink)
     * image_size[1])
147                 target_size = (image_size[0] - particle[4] * (image_size[0] -
    end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))
148 
149                 # Update rotation
150                 rotation = (self._rotation if particle[3][0] <= 0 else -self.
    _rotation) * particle[4]
151 
152                 updated_image = pygame.transform.scale(pygame.transform.rotate(
    particle[1], rotation), target_size)
153 
154             elif isinstance(particle[1], list): # Particle is a spark
155                 # Update size
```

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

1.4.3 Widget Bases

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

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

```
1  import pygame
2  from data.constants import SCREEN_SIZE
3  from data.managers.theme import theme
4  from data.assets import DEFAULT_FONT
5 
6  DEFAULT_SURFACE_SIZE = SCREEN_SIZE
7  REQUIRED_KWARGS = ['relative_position', 'relative_size']
8 
9  class _Widget(pygame.sprite.Sprite):
10     def __init__(self, **kwargs):
11         """
12         Every widget has the following attributes:
13 
14         surface (pygame.Surface): The surface the widget is drawn on.
15         raw_surface_size (tuple[int, int]): The initial size of the window screen,
       remains constant.
```

```
16          parent (_Widget, optional): The parent widget position and size is relative to.
17
18          Relative to current surface:
19          relative_position (tuple[float, float]): The position of the widget relative to its surface.
20          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
```

```
    def set_geometry(self):
        """
        Sets the position and size of the widget.
        """
        self.rect = self.image.get_rect()

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

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

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

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

        Args:
            event (pygame.event.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.
        """
        self._head = None

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

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

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

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

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

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

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

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

            self._head = new_node

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

        Args:
            data: The data to insert.
```

```
        """
        new_node = Node(data)

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

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

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

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

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

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

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

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

        Args:
            data: The data to delete.

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

        current_node = self._head
```

```
128
129         while current_node.data != data:
130             current_node = current_node.next
131
132             if current_node == self._head:
133                 raise ValueError('Data not found in circular linked list! (
    CircularLinkedList.delete)')
134
135         if self._head.next == self._head:
136             self._head = None
137         else:
138             current_node.previous.next = current_node.next
139             current_node.next.previous = current_node.previous
140
141     def data_in_list(self, data):
142         """
143         Checks if the specified data is in the circular linked list.
144
145         Args:
146             data: The data to check.
147
148         Returns:
149             bool: True if the data is in the list, False otherwise.
150         """
151         if self._head is None:
152             return False
153
154         current_node = self._head
155         while True:
156             if current_node.data == data:
157                 return True
158
159             current_node = current_node.next
160             if current_node == self._head:
161                 return False
162
163     def shift_head(self):
164         """
165         Shifts the head of the circular linked list to the next node.
166         """
167         self._head = self._head.next
168
169     def unshift_head(self):
170         """
171         Shifts the head of the circular linked list to the previous node.
172         """
173         self._head = self._head.previous
174
175     def get_head(self):
176         """
177         Gets the head node of the circular linked list.
178
179         Returns:
180             Node: The head node.
181         """
182         return self._head
```

1.4.4 Widgets

Each state contains a `WIDGET_DICT` map, which contains and initialises each widget with their own attributes, and provides references to run methods on them in the state code. Each `WIDGET_DICT` is passed into a `WidgetGroup` object, which is responsible for drawing, resizing and handling all widgets for the current state.

Below is a list of all the widgets I have implemented:

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

1.5 Game

1.5.1 Database

1.6 Shaders