## 1 Technical Solution

### 1.1 File Tree Diagram

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

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

Figure 1: 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`

```python
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.

```

```python
    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)`

```python
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
```

```

def get_angle_between_vectors(u, v, deg=True):
    """
    Uses the dot product formula to find the angle between two vectors.

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

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

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

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

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

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

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

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

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

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

    Returns:
        tuple[int, int]: Vector between the two points.
    """
    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):
```

```python
        # 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)

```python
import json
from pathlib import Path

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

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

    Raises:
        Exception: Invalid file.

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

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

def get_user_settings():
    return load_json(user_file_path)

def get_default_settings():
    return load_json(default_file_path)
```

```

def get_themes():
    return load_json(themes_file_path)

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

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

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

widget_helpers.py (Files used for creating widgets)

```
import pygame
from math import sqrt

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

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

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

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

    return gradient_surface

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

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

    Returns:
```

```python
        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.
    """
    thumb_surface = pygame.Surface((radius * 2, radius * 2), pygame.SRCALPHA)
    pygame.draw.circle(thumb_surface, border_colour, (radius, radius), radius,
    width=int(border_width))
    pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius -
    border_width))

    return thumb_surface

def create_square_gradient(side_length, colour):
    """
    Creates a square gradient for the colour picker widget, gradient transitioning
     between saturation and value.
    Uses smoothscale to blend between colour values for individual pixels.

    Args:
        side_length (float): Length of a square side.
        colour (pygame.Color): Colour with desired hue value.

    Returns:
        pygame.Surface: Square gradient surface.
    """
    square_surface = pygame.Surface((side_length, side_length))

    mix_1 = pygame.Surface((1, 2))
    mix_1.fill((255, 255, 255))
    mix_1.set_at((0, 1), (0, 0, 0))
    mix_1 = pygame.transform.smoothscale(mix_1, (side_length, side_length))

    hue = colour.hsva[0]
    saturated_rgb = pygame.Color(0)
    saturated_rgb.hsva = (hue, 100, 100)

    mix_2 = pygame.Surface((2, 1))
    mix_2.fill((255, 255, 255))
    mix_2.set_at((1, 0), saturated_rgb)
    mix_2 = pygame.transform.smoothscale(mix_2,(side_length, side_length))

    mix_1.blit(mix_2, (0, 0), special_flags=pygame.BLEND_MULT)

    square_surface.blit(mix_1, (0, 0))

    return square_surface

def create_switch(size, colour):
    """
    Creates surface for switch toggle widget.

    Args:
        size (list[int, int]): Image size.
        colour (pygame.Color): Fill colour.

    Returns:
        pygame.Surface: Switch surface.
    """
    switch_surface = pygame.Surface((size[0], size[1]), pygame.SRCALPHA)
    pygame.draw.rect(switch_surface, colour, (0, 0, size[0], size[1]),
    border_radius=int(size[1] / 2))

    return switch_surface

def create_text_box(size, border_width, colours):
    """
    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):
```

```python
        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

```python
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

    def add_laser(self, laser_result, laser_colour):
        """
        Adds a laser to the board.

        Args:
            laser_result (Laser): Laser class instance containing laser trajectory info.
            laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
        """
        laser_path = laser_result.laser_path.copy()
        laser_types = [LaserType.END]
        # List of angles in degree to rotate the laser image surface when drawn
        laser_rotation = [laser_path[0][1]]
        laser_lights = []

        # Iterates through every square laser passes through
        for i in range(1, len(laser_path)):
            previous_direction = laser_path[i-1][1]
            current_coords, current_direction = laser_path[i]

            if current_direction == previous_direction:
                laser_types.append(LaserType.STRAIGHT)
                laser_rotation.append(current_direction)
            elif current_direction == previous_direction.get_clockwise():
                laser_types.append(LaserType.CORNER)
                laser_rotation.append(current_direction)
            elif current_direction == previous_direction.get_anticlockwise():
                laser_types.append(LaserType.CORNER)
                laser_rotation.append(current_direction.get_anticlockwise())

            # Adds a shader ray effect on the first and last square of the laser trajectory
            if i in [1, len(laser_path) - 1]:
                abs_position = coords_to_screen_pos(current_coords, self._board_position, self._square_size)
                laser_lights.append([
                    (abs_position[0] / window.size[0], abs_position[1] / window.size[1]),
                    0.5,
                    (0, 0, 255) if laser_colour == Colour.BLUE else (255, 0, 0),
                ])

        # Sets end laser draw type if laser hits a piece
        if laser_result.hit_square_bitboard != EMPTY_BB:
            laser_types[-1] = LaserType.END
            laser_path[-1] = (laser_path[-1][0], laser_path[-2][1].get_opposite())
            laser_rotation[-1] = laser_path[-2][1].get_opposite()

            audio.play_sfx(SFX['piece_destroy'])

        laser_path = [(coords, rotation, type) for (coords, dir), rotation, type in zip(laser_path, laser_rotation, laser_types)]
        self._laser_lists.append((laser_path, laser_colour))

        window.clear_effect(ShaderType.RAYS)
        window.set_effect(ShaderType.RAYS, lights=laser_lights)
        animation.set_timer(1000, self.remove_laser)

        audio.play_sfx(SFX['laser_1'])
        audio.play_sfx(SFX['laser_2'])

    def remove_laser(self):
        """
        Removes a laser from the board.
        """
        self._laser_lists.pop(0)

        if len(self._laser_lists) == 0:
            window.clear_effect(ShaderType.RAYS)

    def draw_laser(self, screen, laser_list, glow=True):
        """
        Draws every laser on the screen.

        Args:
            screen (pygame.Surface): The screen to draw on.
            laser_list (list): The list of laser segments to draw.
            glow (bool, optional): Whether to draw a glow effect. Defaults to True.
        """
        laser_path, laser_colour = laser_list
        laser_list = []
        glow_list = []

        for coords, rotation, type in laser_path:
            square_x, square_y = coords_to_screen_pos(coords, self._board_position, self._square_size)

            image = GRAPHICS[type_to_image[type][laser_colour]]
            rotated_image = pygame.transform.rotate(image, rotation.to_angle())
            scaled_image = pygame.transform.scale(rotated_image, (self._square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating black lines
            laser_list.append((scaled_image, (square_x, square_y)))

            # Scales up the laser image surface as a glow surface
            scaled_glow = pygame.transform.scale(rotated_image, (self._square_size * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
            offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
            glow_list.append((scaled_glow, (square_x - offset, square_y - offset)))

        # Scaled glow surfaces drawn on top with the RGB_ADD blend mode
        if glow:
            screen.fblits(glow_list, pygame.BLEND_RGB_ADD)

        screen.blits(laser_list)

    def draw(self, screen):
        """
        Draws all lasers on the screen.

        Args:
            screen (pygame.Surface): The screen to draw on.
        """
        for laser_list in self._laser_lists:
            self.draw_laser(screen, laser_list)

    def handle_resize(self, board_position, board_size):
        """
        Handles resizing of the board.

        Args:
            board_position (tuple[int, int]): The new position of the board.
```

```python
            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`

```python
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]
            self._particles.append([None, [radius, random_colour], [*position], velocity, 0])

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

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

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

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

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

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

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

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

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

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

                updated_image = pygame.transform.scale(pygame.transform.rotate(particle[1], rotation), target_size)

            elif isinstance(particle[1], list): # Particle is a spark
                # Update size
```

```python
                end_radius = (1 - self._shrink) * particle[1][0]
                target_radius = particle[1][0] - particle[4] * (particle[1][0] - end_radius)

                updated_image = pygame.Surface((target_radius * 2, target_radius * 2), pygame.SRCALPHA)
                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`

```python
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.

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

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

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

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

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

        self._relative_font_size = None # Set in subclass

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

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

        self._anchor_x = kwargs.get('anchor_x') or 'left'
        self._anchor_y = kwargs.get('anchor_y') or 'top'
        self._fixed_position = kwargs.get('fixed_position')
        scale_mode = kwargs.get('scale_mode') or 'both'

        if kwargs.get('relative_size'):
            match scale_mode:
                case 'height':
                    self._relative_size = kwargs.get('relative_size')
```

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

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

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

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

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

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

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

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

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

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

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

    @property
    def position(self):
        """
        Gets the position of the widget.
        Accounts for fixed position attribute, where widget is positioned in
```

```
pixels regardless of screen size.
        Acounts for anchor direction, where position attribute is calculated
relative to one side of the screen.

        Returns:
            tuple[int, int]: The position of the widget.
        """
        x, y = None, None
        if self._fixed_position:
            x, y = self._fixed_position
        if x is None:
            x = self._relative_position[0] * self.surface_size[0]
        if y is None:
            y = self._relative_position[1] * self.surface_size[1]

        if self._anchor_x == 'left':
            x = x
        elif self._anchor_x == 'right':
            x = self.surface_size[0] - x - self.size[0]
        elif self._anchor_x == 'center':
            x = (self.surface_size[0] / 2 - self.size[0] / 2) + x

        if self._anchor_y == 'top':
            y = y
        elif self._anchor_y == 'bottom':
            y = self.surface_size[1] - y - self.size[1]
        elif self._anchor_y == 'center':
            y = (self.surface_size[1] / 2 - self.size[1] / 2) + y

        # Position widget relative to parent, if exists.
        if self._parent:
            return (x + self._parent.position[0], y + self._parent.position[1])
        else:
            return (x, y)

    @property
    def size(self):
        return (self._relative_size[0] * self.surface_size[1], self._relative_size
[1] * self.surface_size[1])

    @property
    def margin(self):
        return self._relative_margin * self._raw_surface_size[1]

    @property
    def border_width(self):
        return self._relative_border_width * self._raw_surface_size[1]

    @property
    def border_radius(self):
        return self._relative_border_radius * self._raw_surface_size[1]

    @property
    def font_size(self):
        return self._relative_font_size * self.surface_size[1]

    def set_image(self):
        """
        Abstract method to draw widget.
        """
        raise NotImplementedError

```

```python
    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`

```python
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`

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

```

```python
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

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

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

        if self._head.next == self._head:
            self._head = None
        else:
            current_node.previous.next = current_node.next
            current_node.next.previous = current_node.previous

    def data_in_list(self, data):
        """
        Checks if the specified data is in the circular linked list.

        Args:
            data: The data to check.

        Returns:
            bool: True if the data is in the list, False otherwise.
        """
        if self._head is None:
            return False

        current_node = self._head
        while True:
            if current_node.data == data:
                return True

            current_node = current_node.next
            if current_node == self._head:
                return False

    def shift_head(self):
        """
        Shifts the head of the circular linked list to the next node.
        """
        self._head = self._head.next

    def unshift_head(self):
        """
        Shifts the head of the circular linked list to the previous node.
        """
        self._head = self._head.previous

    def get_head(self):
        """
        Gets the head node of the circular linked list.

        Returns:
            Node: The head node.
        """
        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`

```python
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)")

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

            return event_cls(event_type, **kwargs)

        else:
            return event_cls(event_type)
```

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`

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

class ReactiveIconButton(ReactiveButton):
    def __init__(self, base_icon, hover_icon, press_icon, **kwargs):
        # Composition is used here, to initialise the Icon widgets for each widget
    state
        widgets_dict = {
            WidgetState.BASE: Icon(
```

```python
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=base_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.HOVER: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=hover_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            ),
            WidgetState.PRESS: Icon(
                parent=kwargs.get('parent'),
                relative_size=kwargs.get('relative_size'),
                relative_position=(0, 0),
                icon=press_icon,
                fill_colour=(0, 0, 0, 0),
                border_width=0,
                margin=0,
                fit_icon=True,
            )
        }

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

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

reactive_button.py

```python
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): Event to process.

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

```python
        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`

```python
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:
            position (list[int, int]): Position relative to widget rect.

        Returns:
            list[int, int]: Position relative to window screen.
        """
        relative_x, relative_y = position
        return (relative_x + self.position[0], relative_y + self.position[1])

    def set_colour(self, new_colour):
        """
        Sets selected_percent based on the new colour's hue.

        Args:
            new_colour (pygame.Color): New slider colour.
        """
        colour = pygame.Color(new_colour)
        hue = colour.hsva[0]
        self._selected_percent = hue / 360
        self.set_image()

    def set_image(self):
        """
        Draws colour slider to widget image.
        """
        # Scales initalised gradient surface instead of redrawing it everytime set_image is called
        gradient_scaled = smoothscale_and_cache(self._gradient_surface, self.gradient_size)

        self.image = pygame.transform.scale(self._empty_surface, (self.size))
        self.image.blit(gradient_scaled, self.gradient_position)

        # Resets thumb colour, image and position, then draws it to the widget image
        self._thumb.initialise_new_colours(self.selected_colour)
        self._thumb.set_surface(radius=self.size[1] / 2, border_width=self.border_width)
        self._thumb.set_position(self.relative_to_global_position((self.thumb_position[0], self.thumb_position[1])))

        thumb_surface = self._thumb.get_surface()
        self.image.blit(thumb_surface, self.thumb_position)

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

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

        Returns:
            pygame.Color: Current colour slider is displaying.
        """
        if event.type not in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONDOWN, pygame.
    MOUSEBUTTONUP]:
            return

        # Gets widget state before and after event is processed by slider thumb
        before_state = self._thumb.state
        self._thumb.process_event(event)
        after_state = self._thumb.state

        # If widget state changes (e.g. hovered -> pressed), redraw widget
        if before_state != after_state:
            self.set_image()

        if event.type == pygame.MOUSEMOTION:
            if self._thumb.state == WidgetState.PRESS:
                # Recalculates slider colour based on mouse position change
                selected_percent = self.calculate_gradient_percent(event.pos)
                self._last_mouse_x = event.pos[0]

                if selected_percent is not None:
                    self._selected_percent = selected_percent

                    return self.selected_colour

        if event.type == pygame.MOUSEBUTTONUP:
            # When user stops scrolling, return new slider colour
            self._last_mouse_x = None
            return self.selected_colour

        if event.type == pygame.MOUSEBUTTONDOWN or before_state != after_state:
            # Redraws widget when slider thumb is hovered or pressed
            return self.selected_colour
```

## 1.5 Game

### 1.5.1 Database

## 1.6 Shaders