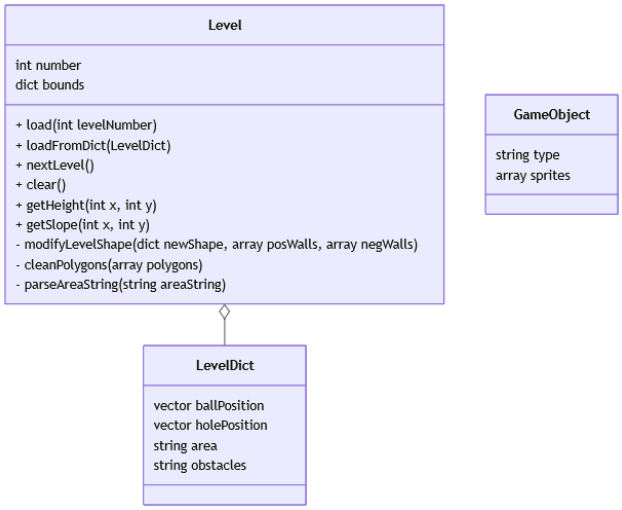
Architectural Document



The process of loading levels in the game involves several complex systems for managing data and displaying content. To accomplish this, we created a level object which handled the loading, rendering, and clearing of levels and their obstacles. Level data is stored in a dictionary, which contains the starting position of the ball, the position of the hole, and a string of code to represent the level area and obstacles.

Defining the play area

Each level has a finite playable area defined by a set of positive and negative regions. Positive regions define playable areas, while negative regions define holes in the positive areas. These regions can form a single contiguous area or can form multiple unconnected islands. The shapes of these regions are created using an in-house scripting language that allows our level designers to describe a series of boolean operations on primitive shapes to easily create more complex geometries.

```
// Right side;
{
    ADD rect 350, 0, 150, 250;
    SUB {
        ADD circle 390, 165, 40;
        ADD rect 350, 125, 40, 200;
        ADD rect 390, 165, 40, 200;
    }
}
```

The level object takes this string of commands and parses it one at a time, adding and subtracting geometry until the final shape is complete. Rectangular colliders are placed along the edges of each resulting polygon to keep the ball within the space. Upon clearing the level, all data relating to the level's shape is reset and all walls are deleted.

Drawing the play area

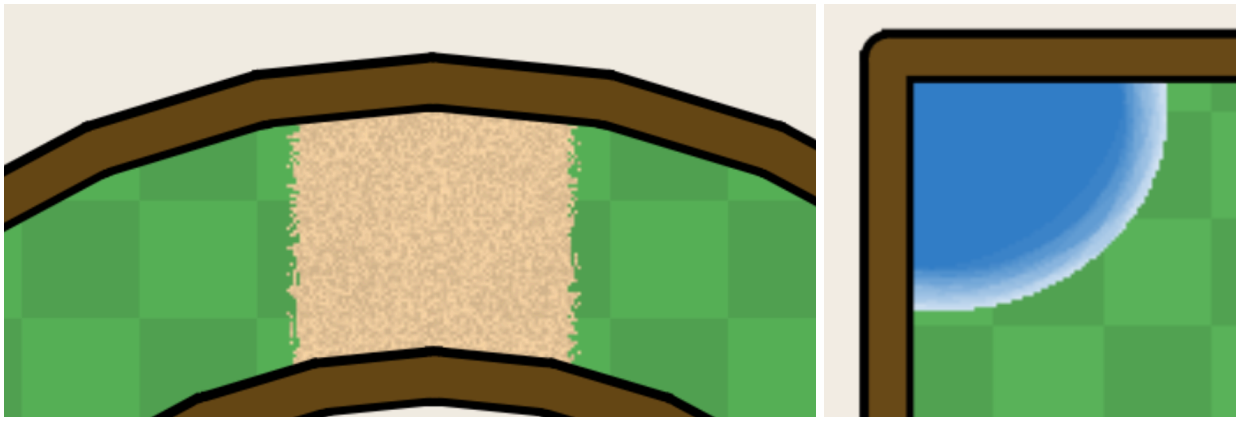
Once the play area is defined, the polygons are drawn using a WebGL shader to produce procedural effects on the ground. A checkerboard pattern is created by rounding the pixel coordinates to a specific interval and darkening the ground if the sum

of the x and y coordinates equals an even number. Shading is also used to represent elevation, with lighter regions representing higher elevations and darker regions representing lower elevations.

Rendering sand involved using a noise function to generate a random value to shade each pixel by. This created a nice grainy texture. To make the edges of sandy regions look a little messier the shader looks at the ground surrounding the sandy pixels. If any ground around the pixel is not sand, the pixel is marked as being on the edge and has a small chance to be rendered as grass instead. This made the edges of the sand look much rougher and gave a more realistic appearance.

Water is primarily rendered as a plain blue texture, but with white waves where the water meets grass. The same edge detection technique as before is combined with a sine wave to make the waves oscillate.

Checkerboard pattern with an elevated hill on the left side.



Sand and water; note the details where each region meets the grass.

Handling game objects

Game objects consist of one or more P5play sprites that are given special behavior. Since our game is single threaded, all sprite behavior needs to be handled in one sequence. We accomplish this by loading all game objects into a single array, then each frame iterating through that array to execute each sprite's behavior.

All game objects are instances of a single GameObject class, which contains a type specifier and an array of sprites. To execute a game object's behavior, it checks its type and performs the relevant action. To unload the level's obstacles, we iterate through the game objects array and call a destructor function on each item, which deletes all the object's sprites.

Menu management

The final system used in the game is not for the game at all, but rather its menus. All menu interfaces are designed as HTML elements and made to be children of a single menu element, which is in turn a child of the menu manager element. We have a single variable that dictates the current menu, and, when it is changed, all children of the menu manager are updated so that only the currently selected menu is visible.