Developers guide:

Overview

Our entry point is index.html, which gives us a couple things: the styling of the game canvas (middle of the screen), and the achievements below the canvas.

Then we enter the GameScene. This is an extension of phaser scene and allows us to do all the drawing logic, handle inputs, time events, update logic, tweens, etc. It is very useful and necessary for this project, as many parts of our project depend on the scene.

The core part of the actual gameplay is found in the Level class. It handles the interactions of shapes, and spawns them. We also made it so that the end of the level is when there are 0 shapes left, so then we get GameScene to get new level info. Important to note, the level can only end if there are 0 shapes left. I made it so the shapes despawn if they go 50 pixels below the canvas. Some shapes go upwards, however you'll see that when they go off screen, I artificially give them a large downward velocity when the level should end, so that they despawn correctly (see levels.js -> level8 for an example).

The main interactable object of this game is the Shape class, through which everything that the player can interact with. It uses the graphics object to tell if the user's mouse interacts with it. I chose this object since there is a handy "pointerover" method that is made with graphics objects in mind.

Essentially, any object that is moused over will have some interaction on the game state.

The types of shapes are Squares, Triangles, ArmoredTriangles, StealthTriangles, Areas, Architecture, and Upgrade, which itself has ClearFriendlies, SlowTime, and Intangible as sub-classes.

When creating a level, we can use any collection of these shapes to present the player with challenge. Levels.js contains the text data of the shapes to use when creating levels. There are a few properties to all the shapes (except architecture) that we can play with:

1) x is the beginning x position, and y is the beginning y position.
2) velocityX is the beginning x velocity and velocityY is the beginning y velocity.
3) angularVel is the rate at which the shape spins

There are also 4 ways to affect the movement in more complex ways:

1) newVelocity changes the velocityX and velocityY over certain time parameters in the form of

[[timeDelay, [newVelocityX, newVelocityY, durationOfChange], etc]
2) growing changes the size of the shape in the form of
[[timeDelay, [durationOfChange, newScale], etc]
3) linearMovementData is a back and forth movement that can repeat in the form of
[[timeDelay, [durationOfMovement, repeatXTimes, xVel, yVel], etc]
4) circularMovementData allows for continuous circular paths in the form of
[timeDelay, duration, radius, speed, startAngel]

Additionally, there are a few special cases of shapes:

1) Architecture takes an array of points in the form
[{x: num, y: num}, ...] to define the shape, and a downward velocity.
2) Black turns the screen black for a certain period of time and takes data in the form of
[delayBeforeStart, durationOfFade, repeatXTimes]. It also holds the black screen for 250 seconds (regardless of difficulty), and has 1 second between screen blackening. If you would like to adjust that so it's harder or easier on different difficulties, it is a simple manner of adjusting like the other animate methods in shape, however I found this more fun upon playtesting.

With these properties, it is possible to have complex movement patterns and interesting shapes to create difficult levels

The levels.js contains all the level data. It allows for size changes, and a multitude of movement changes. In the future, we can make extensions to include the type of tween movement (Sine.easeIn, Linear, etc), and allow architecture to allow movement as well.

One thing that may be difficult to understand is the difficulty adjustment when we are calculating the deltaX and deltaY, it doesn't multiply the secondVelocity or circularVelocity, illustrated below:

```
const deltaX = ((this.velocityX * this.scene.difficultyAdjustment +
this.secondVelocityX + this.circularVelocityX) * delta *
this.scene.slowtimeFactor) / 1000;
```

This is because the animate*** methods already take the difficulty adjustment into account when making the tweens, and it would double dip on the adjustments otherwise

Most of the classes regarding shapes, like the triangle, square, upgrade, stealth_triangle, armored_triangle, and area, are mostly the same, with just variations on shape to draw correctly, and then the Level class is the one that handles their unique interaction when moused over with the cursor. handleShapeInteraction() does most of the work in the regard, depending on

the shapeType. Other parts of the code handle interactions as well, such as update, by checking if the shape is an instanceof a specific class.

There are various menus in the game, and the cutscene. I opted to keep it simple and just show and hide the various menus and images/sentences based the logic of what is clicked.

If more details about the code are needed and how to use them, check the documentation in the Docs folder (made using JSDocs)