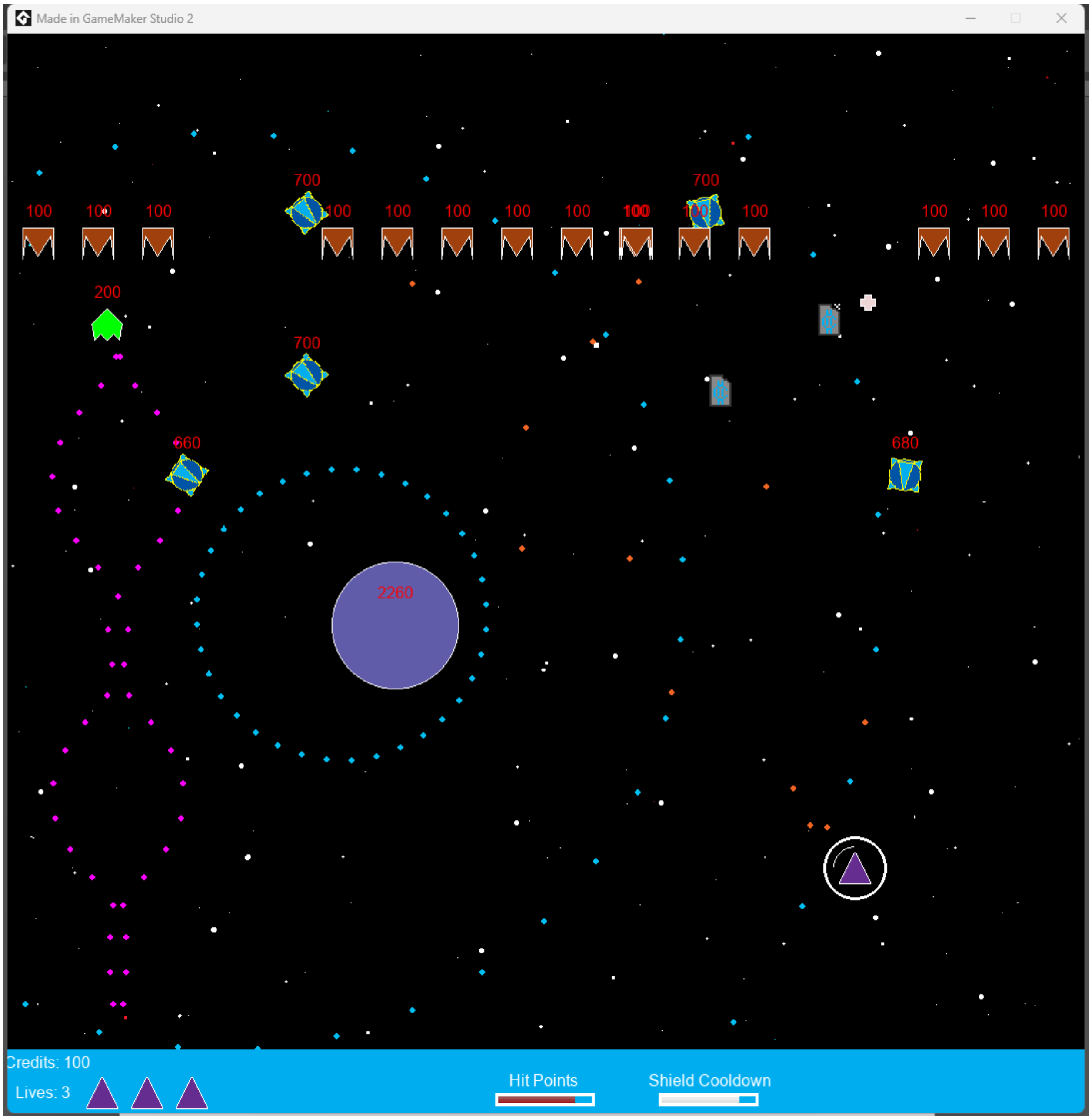
Polygon Pilots Documentation

Milestone 2
Trevor Caserio
CS 3540

Game Design

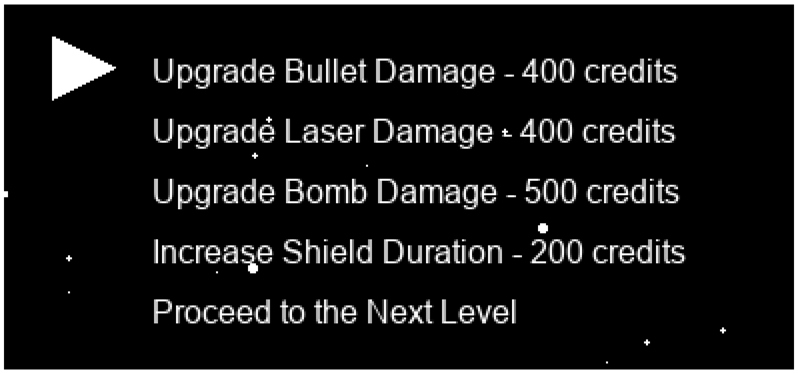
The game I envisioned was a bullet hell arcade shooter. The objective of the game is to survive and clear waves of enemies. Completing the five levels of the game earns the player a victory. Losing the three lives the player has will result in a game over.



Example Gameplay - Player is the purple triangle on the bottom right

The enemies of the game all had to vary in their attacks and behavior. I wanted a player to see an enemy appear on the screen and have to make in the moment decisions on how they will change their strategy to survive and eliminate the new threats. When the bomber enemies spawn that track towards the player, the player will have to adjust by evading, shielding, or prioritizing destroying them. The goal is that all enemies will trigger this reaction in the player so that they are always adjusting their strategy based on what they've learned about the enemies. This creates a lot of tension and moment to moment decision making.

Enemies might drop credits for the player to pick up. Picking them up add to the player's total credits that can be spent between levels. Players can use this resource to increase damage to one of their weapons, the bullets, laser, or bomb. Or they can upgrade the duration of their shield. Doing so allows the player to change their playstyle. Do they want to be more reckless and increase the shield time? Do they want to drop bombs and eliminate enemies that way? I wanted to give players ownership over their preferred playstyle while at the same time offering a potential variety between playthroughs.


Upgrade Bullet Damage - 400 credits
Upgrade Laser Damage - 400 credits
Upgrade Bomb Damage - 500 credits
Increase Shield Duration - 200 credits
Proceed to the Next Level

Upgrade Prompt

To summarize, I wanted to create a fast paced bullet hell game with a variety of encounters where the player got to make decisions that influenced their playstyle.

PlayTesting

Visual Clarity was something brought up multiple times in testing. I gave the player's bullets a white outline and made the sprite size bigger to make it clearer where the player is firing.

Another player asked for more offensive options. I added a bomb projectile that made a more unique way of attacking rather than just positioning the player under the enemy.

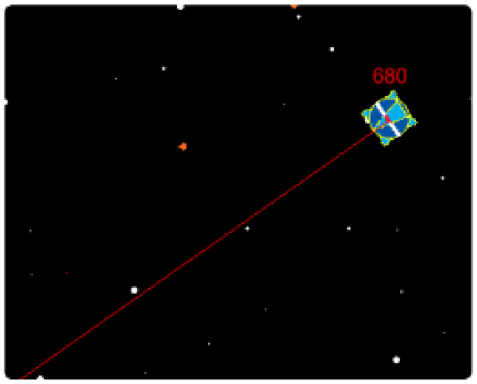
Before I added levels, I had all the enemies in one sequence of spawns. The difficulty ramped up too quickly but I've created the early levels to be easier to ease the player into the game.

Game AI

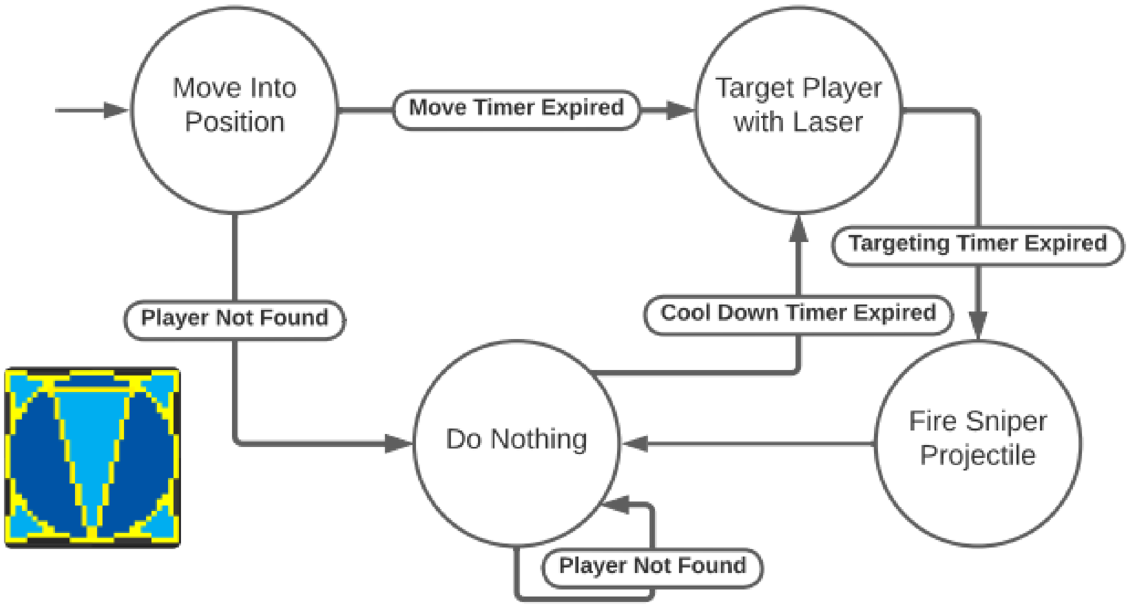
I designed the game to be an enemy and bullet dense obstacle field for the player to navigate. Because of this, I mostly take advantage of unit style AI. Each enemy has a simple AI script that either follows a state machine or reacts to the player. I'll explain how a few of the enemies operate in the game.

Enemy 3 “Sniper”:

The Sniper has 4 states it can be in. First, it’s initial moving state where it moves into position on the screen. Second, It’s targeting state where the sniper targets the player with a laser and the enemy flashes in a way to show it is about to fire. Third it fires a sniper bullet. Fourth, it enters an idle state. A diagram below shows the transitions between states, conditions and times, and some screenshots.

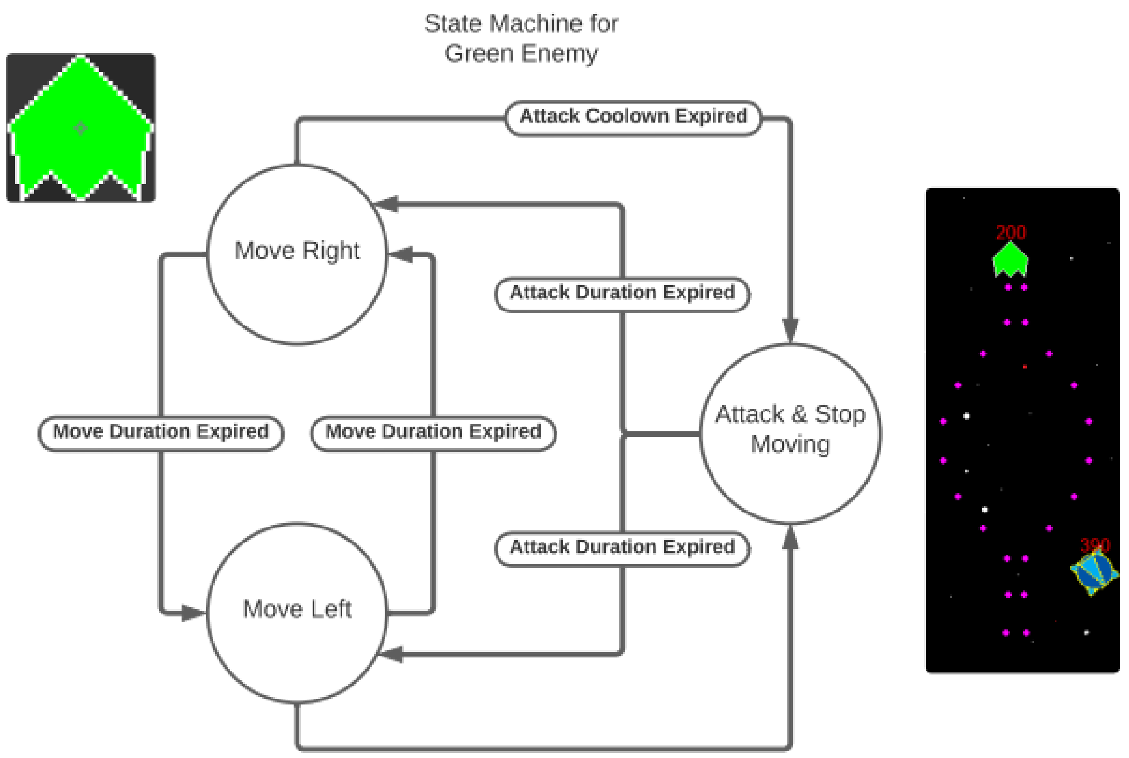


State Machine for Sniper Enemy



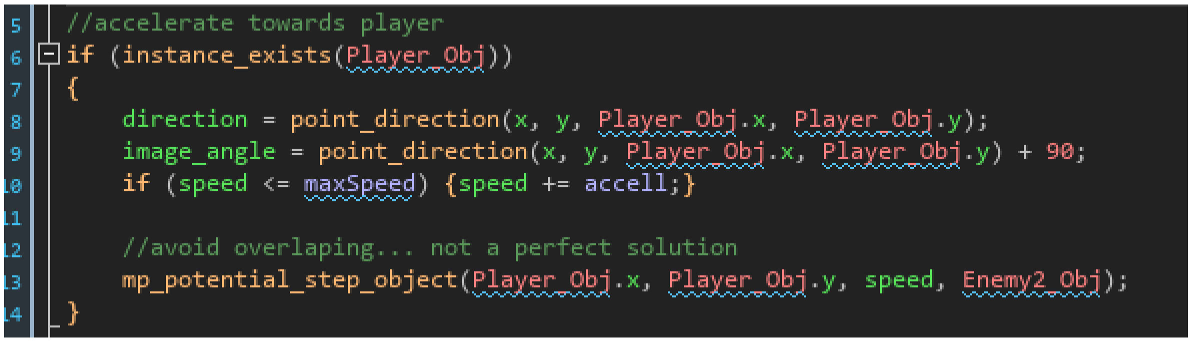
Enemy 4 “Green Enemy”

The Green enemy transitions between moving right and left after spawning based on an internal timer. Another timer transitions the states from a move state to an attack state. At this point the Green enemy stops moving and fires two lines of bullets that oscillate like a sine wave. The idea here was that the enemy was vulnerable when it was in a move state, but while it was in its attack state, the player would either have to take the risk of moving in between the bullet pattern or wait for a safer time to attack. A Diagram is shown below to demonstrate the state transitions.



Enemy 2 "Bomber":

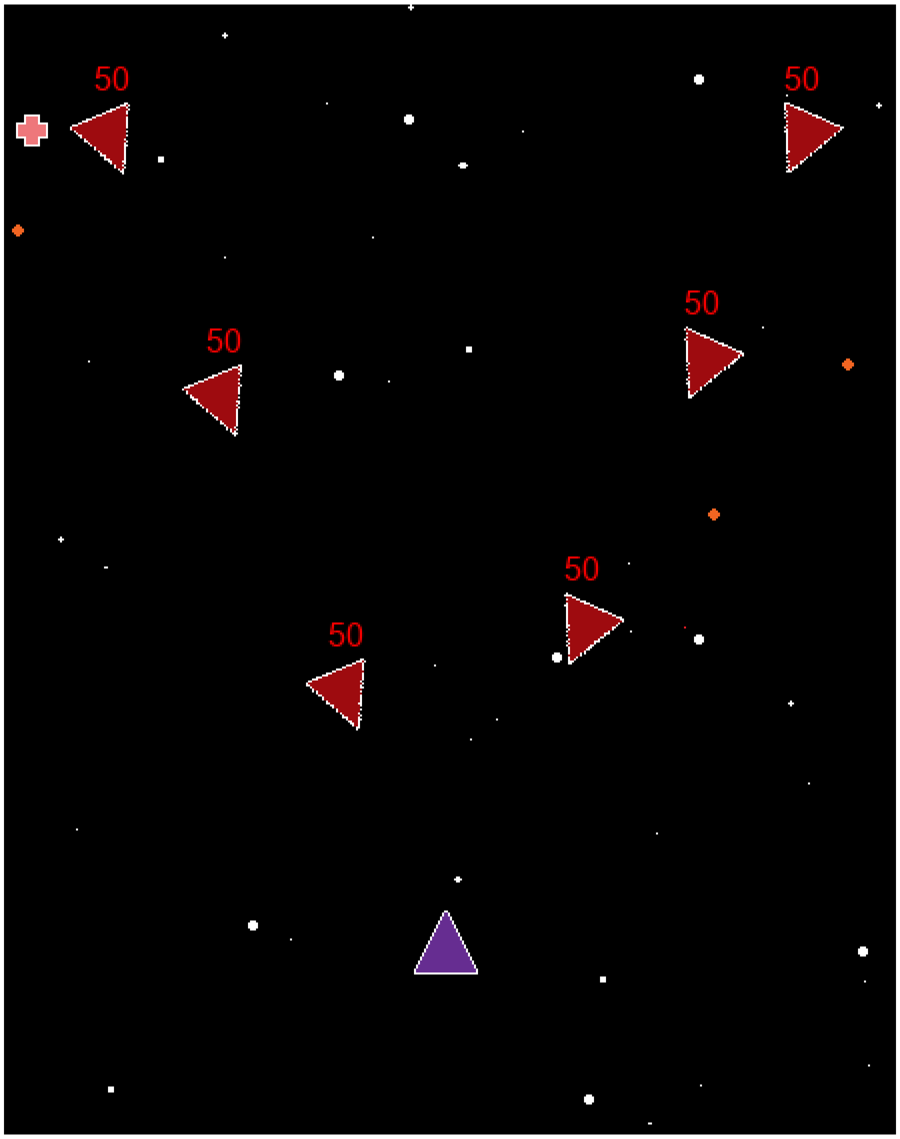
I have a simple enemy that accelerates towards the player and does damage on impact. They typically spawn randomly or in packs to create an immediate threat that the player will need to respond to. It's a simple pathing algorithm, but it does plan it's path based on the players current position. The logic is show in the code below.



```
//accelerate towards player
if (instance_exists(Player_Obj))
{
    direction = point_direction(x, y, Player_Obj.x, Player_Obj.y);
    image_angle = point_direction(x, y, Player_Obj.x, Player_Obj.y) + 90;
    if (speed <= maxSpeed) {speed += accell;}

    //avoid overlaping... not a perfect solution
    mp_potential_step_object(Player_Obj.x, Player_Obj.y, speed, Enemy2_Obj);
}
```

Since this executes every frame, a bomber unit is constantly adjusting its movement direction and sprite angle (image_angle) towards wherever the player is. The speed also increases up to a cap so players need to take them out before they become a problem.



Bomber enemies tracking a player

Other Enemies:

There are 3 other enemy units in the game. Their use of AI is very similar either responding to the player or transitioning between states. The basic fighter moves across the screen and fires towards the player. The Big Circle enemy moves rapidly across the screen, radiating blasts of projectiles or stopping to create a spiral of outwardly moving projectiles. The Boss enemy transitions between raining projectiles in front of itself or recharging its attack while moving across the screen.

I wanted a variety of attack and behavior patterns so that the player could recognize these attacks over time and plan what to attack first, how to move, what attacks to use, etc… based on what threats are on the screen. Even as I test this game, the mastery of defeating these enemies has been a satisfying experience.

Software Design

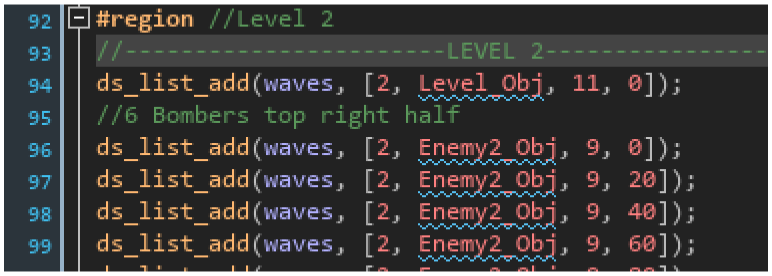
Inheritance/Generalization:

Enemies all inherited from a single Enemy Parent Class. This way when player attacks collide with anything of the Enemy Parent type, it would deal its damage.

Drops had a parent drop object so that any drops would be able to similarly track to the player when they got near.

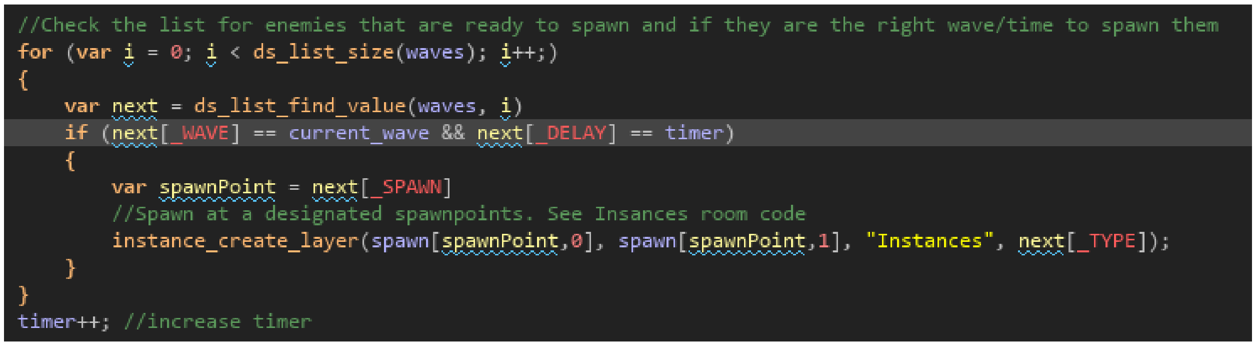
Spawner:

This is probably the most vital part of this project. The spawner is responsible for maintaining spawns and controlling the flow of the game. It spawns the enemies, some UI elements, and keeps track of all of these objects. The Spawner is passed a list of objects, the level they belong to, their spawn points, and spawn time.



```
#region //Level 2
//-----------------------LEVEL 2---------------
ds_list_add(waves, [2, Level_Obj, 11, 0]);
//6 Bombers top right half
ds_list_add(waves, [2, Enemy2_Obj, 9, 0]);
ds_list_add(waves, [2, Enemy2_Obj, 9, 20]);
ds_list_add(waves, [2, Enemy2_Obj, 9, 40]);
ds_list_add(waves, [2, Enemy2_Obj, 9, 60]);
```

This code above is an example of how enemies are added to a level. The array parameter of ds_list_add() contains the [level id, object type, spawn point, spawn time].
The spawner is always checking each frame for objects ready to spawn as shown in the code snippet below.

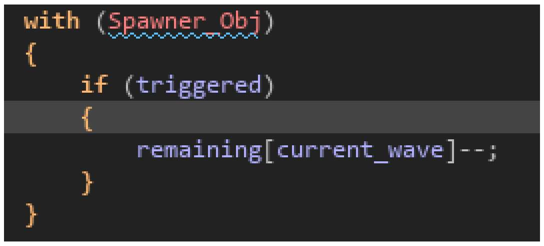


```
//Check the list for enemies that are ready to spawn and if they are the right wave/time to spawn them
for (var i = 0; i < ds_list_size(waves); i++;)
{
    var next = ds_list_find_value(waves, i)
    if (next[_WAVE] == current_wave && next[_DELAY] == timer)
    {
        var spawnPoint = next[_SPAWN]
        //Spawn at a designated spawnpoints. See Insances room code
        instance_create_layer(spawn[spawnPoint,0], spawn[spawnPoint,1], "Instances", next[_TYPE]);
    }
}
timer++; //increase timer
```

Pseudo code for this algorithm:

```
Iterate through each enemy/object for the level:
    If the level id matches the current level and the current time match the spawn
    time:
        Spawn the Enemy at the correct spawn point
Increment the timer
```

The spawner moves to the next level and the next list of enemies to spawn when it sees that there are no more remaining objects. That means that every enemy needs to communicate with the spawner when they are destroyed so that the remaining enemies/objects variable can be updated.

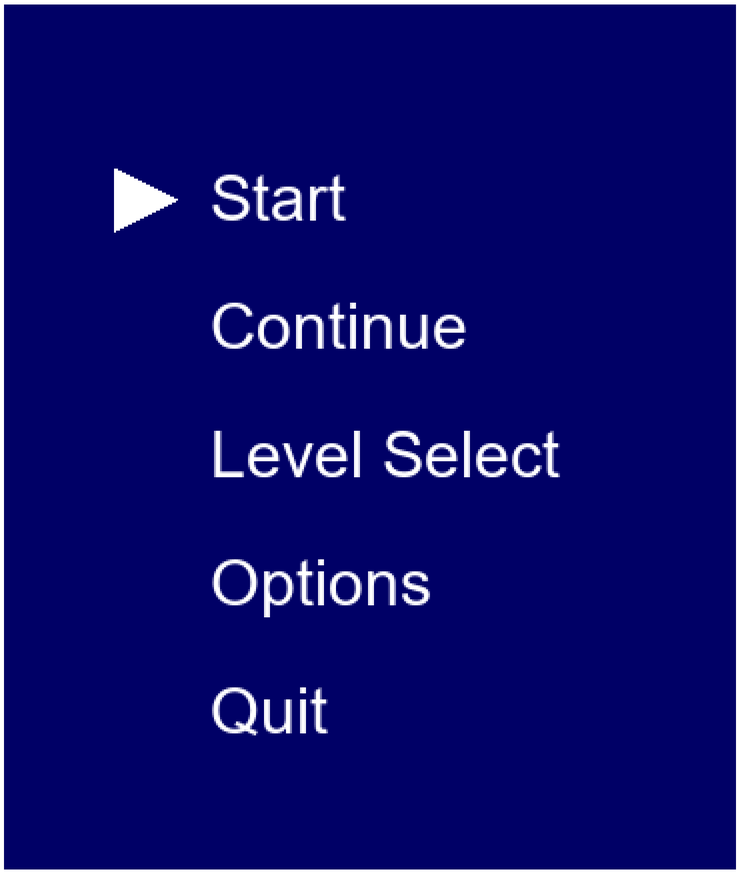


```
with (Spawner_Obj)
{
    if (triggered)
    {
        remaining[current_wave]--;
    }
}
```

When the remaining objects value for that wave is 0, the next level will begin.

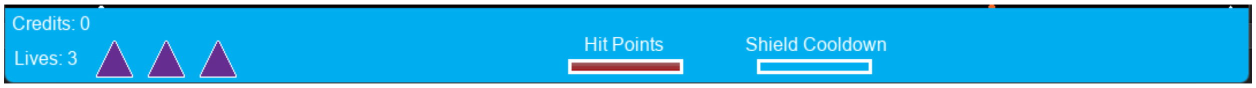
UI Elements

The menus use an array and a case statement to determine the logic of each selected menu item. This is both in the start up menu and the upgrade selection.



Startup Menu

The bottom of the screen contains player information like lives, credits, health, etc.



Player UI

Credits are a global variable so that they can be accessed by the upgrade menu UI object. The other items on that interface are tracking the player object's member variables.