Spaceship Game

Design Overview

Contents

Nguyen, Jeremy Lam

Section 1

How to set up MIPS to play the game

1. **Environment:**

First, download MIPS, then open the game file. This game only has one file, so all it needs is to be assembled as a single file.

After that, go to “tools” on the toolbar and click on “Bitmap Display” to display the Bitmap. Set both the unit width and height in pixels to 4, then set the display width and height in pixels to 256px and 512px, respectively. Make sure that the base address for display is at heap data (0x10040000).

Note that after adjusting the display width and height in pixels to 512, the Bitmap window might not automatically expand, make sure to adjust it to properly see the full Bitmap display.

Click “Connect to MIPS” on the bottom left of the Bitmap window.

Proceed to “tools” on the toolbar and click on “Keyboard and Display MIMO Simulator”, then click on “Connect to MIPS” on the bottom left of the new window.

2. **Assemble and run**

Click “Assemble” then run the program. The program will start.
Select the desired ship, then click on the MMIO keyboard to start registering keystrokes.
No need to reset bitmap display, the program automatically wipes it clean every new play.

3. **Controls**

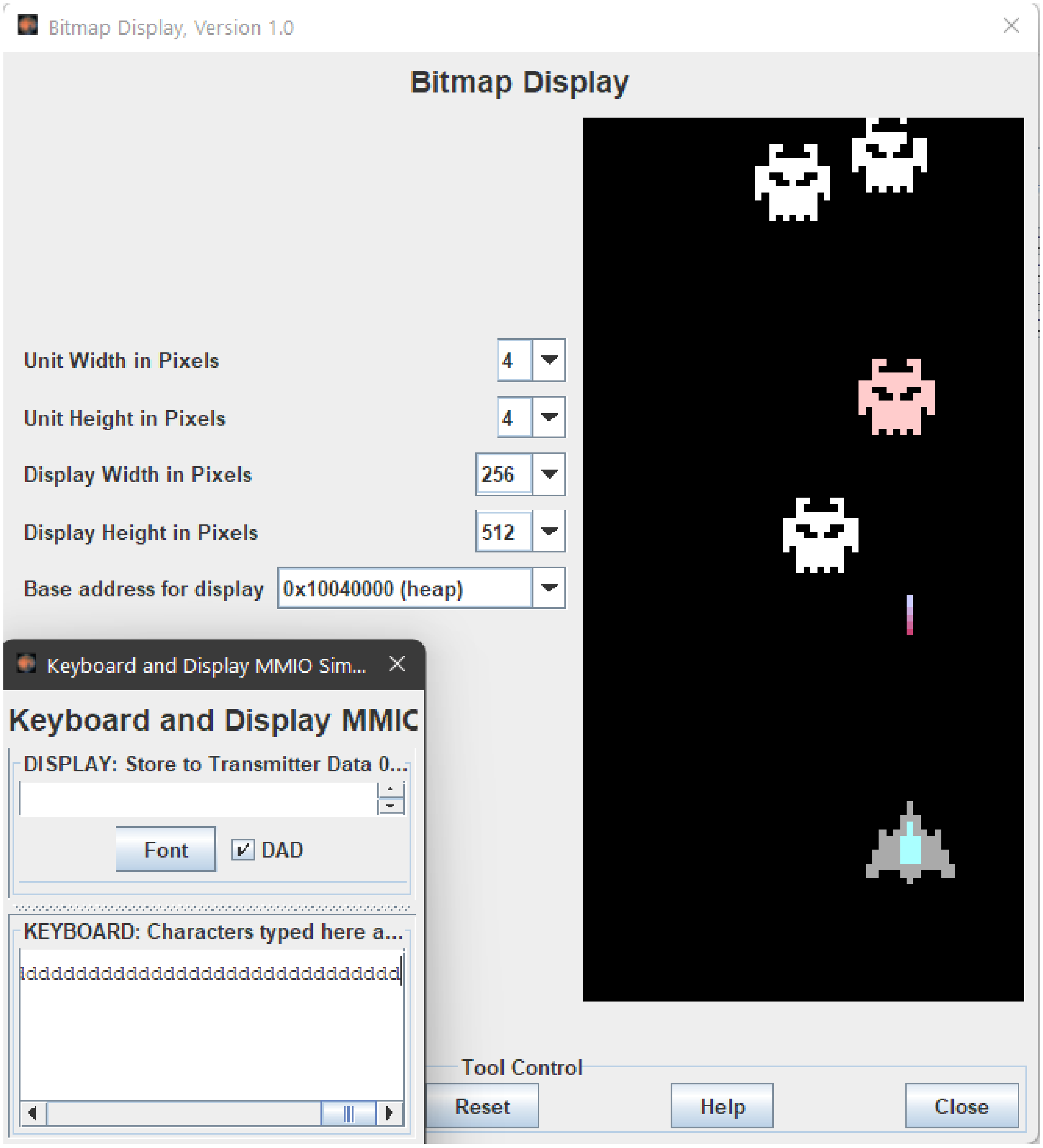
Use W, A, S, D to control the spaceship UP, LEFT, DOWN, RIGHT, respectively.

To move continuously press and hold the desired key.

To exit, press “space”.

Section 2

Game Demo



Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels: 4

Unit Height in Pixels: 4

Display Width in Pixels: 256

Display Height in Pixels: 512

Base address for display: 0x10040000 (heap)

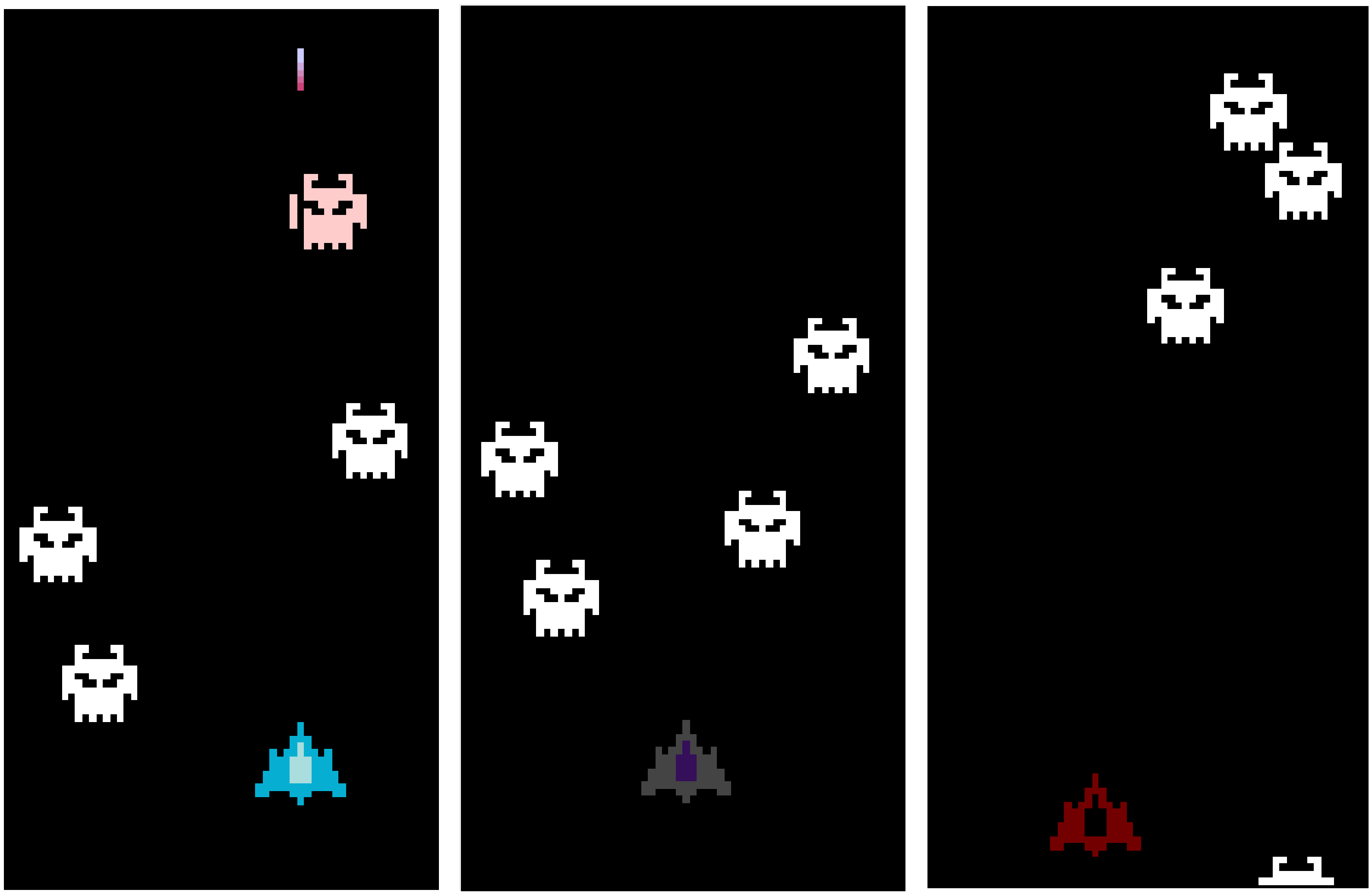
Keyboard and Display MMIO Sim...

Keyboard and Display MMIC

DISPLAY: Store to Transmitter Data 0...

KEYBOARD: Characters typed here a...

```
ldddddddddddddddddddddddddddddddd
```


Actual Game Play

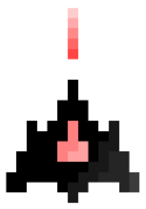
Section 3

Game Design

General Design Description and Motivations

There are two kinds of object categories in this game, the visual objects (Player's ship, laser, enemy's ship, etc.) and the value objects (score, cycle, etc.). A visual object is an array of *.word*, used to store the objects "origin coordinate", color, and other information. For example, the player's ship has its origin at the tip at the very front.

1. Player's ship design



The overall height and width of the player's ship is 12 pixels by 13 pixels, respectively. The odd number of pixels of the width is to ensure symmetric in design and ease in usage in the coding prcess. In game, though, the body of the ship is grey, instead of black (to contrast with the default Bitmap background color). The ship's core is the center piece that is in bright color. This is where the player must protect, as when the core collides with the enemy, the player loses.

In order to make the ship move for every keypress, the program takes an input of the keyboard MIMO, then makes a decision whether to move or not. If it moves, determine the direction by which key is pressed. If no key is pressed, the the ship is stationary. When moving in the x direction, for example, the program blackens the location of where the ship is currently at, then update the ship's X-origin by adding/subtracting, then draw the ship at the new location.

On top of the space ship is its laser projectile, used to shoot its enemies' ship. This laser is a visual object. There is a gradient effect, generated during drawing the shooting animation, which give the laser a more dense, energy-carrying, and burning-hot look. In order to "fire" the laser, the program loads the origin of the ship, which is at the top tip (which is also the weapon), then stores it into the origin of the laser every *cycle* then draws it.

2. Enemy's ship design

Like the player's ship, the enemy's ship is a visual object. The only difference is that it always moves from top downwards, towards the bottom of the screen, and it can only move in a straight line. There are always four enemy ships on the screen.

Their spawning origin is randomly generated using syscall 42. Their downward velocity increases the longer the game is played. This is implemented so that the game gets progressively harder. In addition to that, the enemy could become rouge, they will move side to side, stops, then accelerate downwards. This is used to taunt players. This feature can be turned on or off in the source code, which is noted.

When the enemy is being shot at, they blink red, indicating that they are damaged. When they are destroyed, they will disappear from the screen.

3. Value Objects

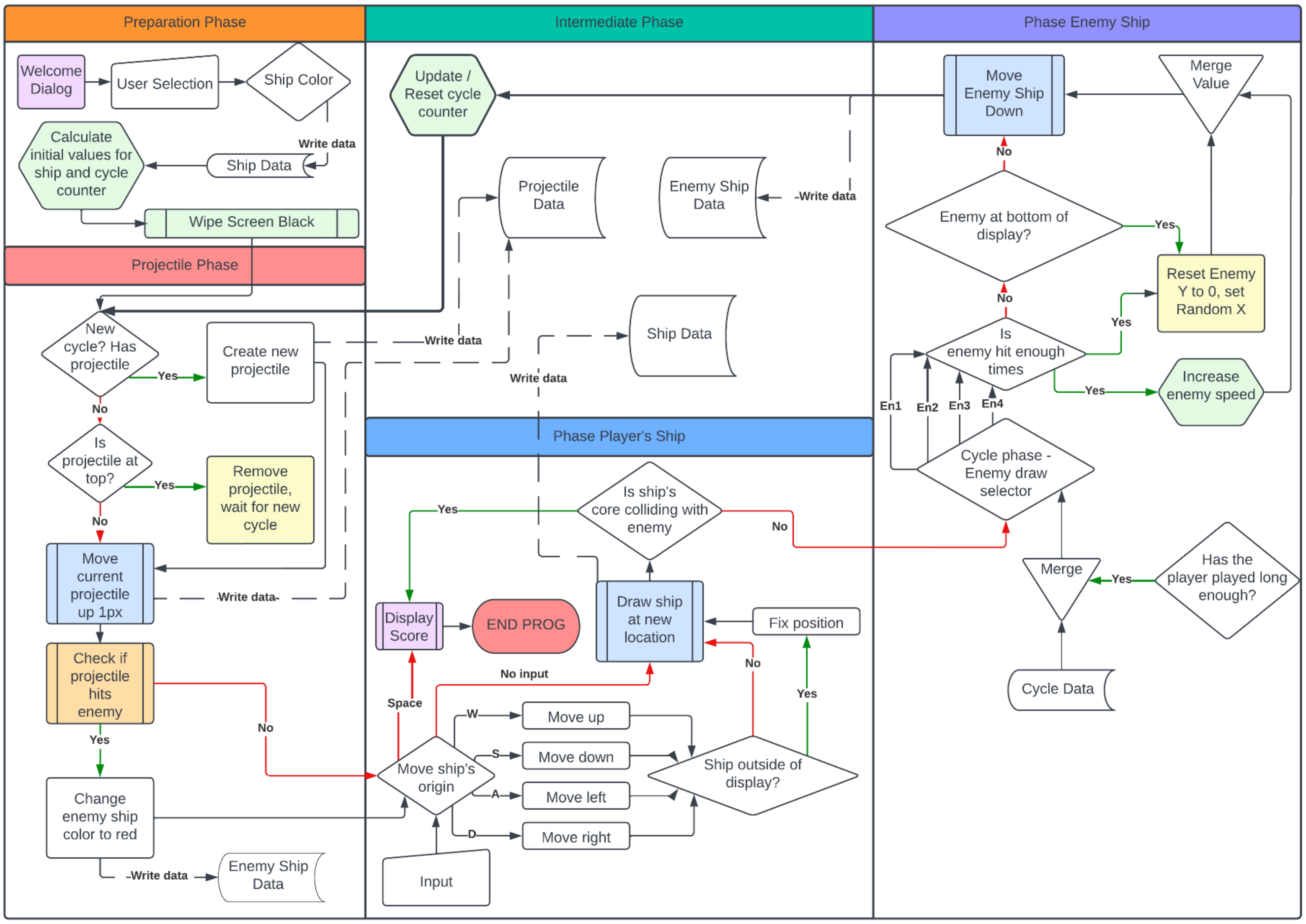
Two units that is used often in this documentation of the game are *cycle* and *cycle counter* (or counter for short). A *counter* can be defined as one full loop of the program and a *cycle* contains 128 *counts* of counter. It takes a pixel up to 128 nonstop single-pixel-movements to travel the height of the display (64x128) only once, so therefore the cycle is defined the way it is in this program.

The player score is calculated by the number of successful hits, where the projectile impacts the enemy ship. These values are counted during program execution and will show once the player exits the program. Real life observation motivates this score system. The reason for this is if the player shoots a little antenna on the enemy ship, then it does not affect the overall ship. But if the player shoots the enemy at their core, or wing, or engine, then it would damage their ship more – so the player should be rewarded more.

Section 4

Program Overview

High Level Overview and Implementation Detail of the Program



Phase Preparation

At start, the program will load a pop-up, welcoming the player. Then it will load the initial origin state of the player's ship. It will also set up the cycle counter, which is going to be used for all moving objects excepts the player's ship. A *cycle* in this program is defined as 128 counts, each *count* is one full program loop.

Phase Projectile

The idea behind the projectile is that for every cycle, the projectile will travel from the ship's origin (which is a weapon system also) to the very top of the screen. At the start of the cycle, the program will load the origin of the ship as the origin of the projectile, this is to ensure that even when the ship moves, the projectile will always be shot out from the weapon of the ship. Then after every *count* the projectile will move up by one pixel. This creates a fast and smooth movement. After the projectile moves up, it checks for if it hits the enemy. If the enemy is hit, the corresponding enemy's object will save *RED* as it's color so that it can be drawn in red, indicating that the projectile has hit it.

Phase Player's Ship

The idea behind the player's ship is that there are 2 functions to draw the ship with colors and to black out the ship. The functions use the origin of the ship as it's starting point then it traverses a set sequence of store to memory and move across locations in the heap. When there is no user input, the ship stays still. When there is an input, the MMIO keyboard will store data to a register so that the program can decide what to do.

For example, if the player presses "A" the program will branch to the A branch. In the A branch, the program first calls the *blackout function* to completely blackout the current ship. The previous origin, say (x, y) = (64, 64) will be updated to (x, y) = (63, 64). This update will be reflected when the *draw function* is called. Then the program calls the *draw function* to draw the ship, which is now translated one unit to the left.

By repeating this process, the ship will look like it is moving, when in reality the ship is continuously being erased, updated, and drawn.

When the ship's core collides with the enemy's ship, the program will branch to *end*. See below.

In the case where the player input "space" the program will branch to *end*. At *end*, the program calculates the score (see section 3.1 for more calculation details) then displays it in a dialog box. If the player presses any other keys, the program will ignore.

Phase Enemy Ship

Enemy ships use the cycle counter to decide when to move. For example, when the count is at 64, the enemy ship #1 will move and the other ones do not. This design is motivated by two reasons. The first reason is so that the enemies' ship moves slow enough for the player to dodge. In practice, each ship moves down twice per cycle, in contrast to the player's ship projectile moving up to 128 times per cycle. The second reason is so that at each count, the program only must delete, update, and draw the enemy ship only once, while splitting it eight times total per cycle. This translates to a smoother experience for the player.

When the enemy ship is shot at, the color of the ship is changed (see *Phase Projectile*) and the *hit counter*, defined in the enemy's ship object, is incremented. This represents the number of hit the object has received. Once the *hit counter* is high enough, the ship will disappear. Another ship would then appear with a higher velocity (if the functionality is turned on, the ship will also do some side movements to taunt the player).

The automatic down movement of the enemy is quite simple. Similar to the design of the player's ship there are *draw* and *blackout* functions for the enemy ship. Every time it is the turn of a ship to move, the program blackouts the current ship, adds the value of the *speed* data, determined in the object's data array, then finally drawn. In the case if the ship is hit by the incoming projectile from the player, the color will change to slightly red. This will be reflected momentarily when the *draw function* for the enemy ship is called.

Once the ship starts to move out of screen, the *draw function* automatically handles the drawing so that the program does not draw outside of the allocated memory. When the ship is completely out of the screen, the program resets the Y origin of the ship, while randomizes the X origin of the ships to create a variety of challenges for the player. There are about 1.6×10^{16} ways the enemy could appear, so the player will get to experience different challenges.

Section 5

Issues

Bugs That Appear During Execution and other Errors

Inconsistent Enemy Respawn

This bug happens randomly. The enemy could be hit only once but would respawn again, instead of being hit 16 times, as defined in the program. There is no fix yet.

Static Memory Overflow

The program used to write the bitmap information to static memory, but that would cause big issues, with artifacts and remnants of objects presented on the screen. It also increases the change of *Inconsistent Enemy Respawn*. It also introduces runtime error that cause the program to halt. A fix to this problem is to move everything to the heap.

Enemies Not Registering Hits

This problem is the opposite of the first problem. The enemy sometimes could take 100 hits from the player and does not disappear and respawn. There is no fix yet.

Enemy Ship is Overridden by Projectile

Since the flow of the program is Projectile – Ship – Enemy, parts of enemy ship is blackened when the projectile hits and stays there until the next time the Enemy ship is drawn. A solution to this problem is by setting the flow as Projectile – Enemy – Ship – Enemy. But the price for this solution is that each *count* takes longer to complete, making the overall program slow.