**CDDS Optimise Design Brief**

CDDS Assessment Task 1 is to optimise the CDDS\_Optimise project in the AIE Student Samples Solution. By optimise, the task will have you either make a performance, or a design improvement.

**Design Improvements:**

The current main file contains all the functionality of the program, which is messy and difficult to read. For a design improvement, I will move all the functionality into its own class and separate functions/methods. This segments the code, allowing for improvements to readability, and easier troubleshooting.

A computer screen shot of a program code

Description automatically generatedAfter creating a Game class – consisting of .h and .cpp files – I searched through main for important variables and added them into the .h file under the “private” header. I then turned the critter objects into pointers, and added raylib.cpp to the project, and used its window object as a pointer. I then added “m\_” to all the variables, which improves clarity for the reader, knowing that those variables come from the header file.

I then added a constructor, destructor, and the Run() method to the “public” header of the class. The Game() constructor will contain all our initialisation, and the Run() method will contain the program’s loop.

A computer screen shot of a computer code

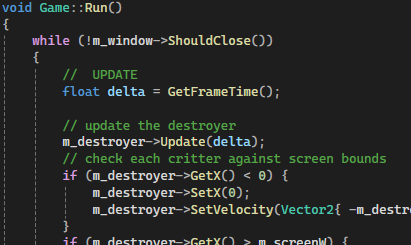
Description automatically generatedI then copied the initialisation of the program into the Game() definition in the .cpp file. And after reformatting the code – because the variables have new names and types – an error occurred because we were trying to initialise our critters, but our array of critter pointers had not been allocated any memory yet. So, I added a line to create critters using the “new” keyword.

A screen shot of a computer code

Description automatically generatedWe had to repeat this same fix for our destroyer object, who also became a pointer.

A computer code with white text

Description automatically generatedBecause our program is now manually assigning memory, we need to also unassign our memory to avoid a memory leak. We use the “delete” keyword in the destructor to deallocate the memory we allocated with the “new” keyword. Because we allocated memory in a loop, we also need to deallocate memory in a loop to not miss anything.

The last step is to move the loop into the Run() function. This also required adding “m\_” to all of the variables and changing the “.” to “->” for all the method calls, because of the variables now being pointers. Additionally, we now also reference the dedicated “m\_window” object for the loop. Picture for reference is not the whole function.

A black background with white text

Description automatically generated**Performance Improvements:**

I created an average FPS calculator in the program that outputs its value once the program has run for 10 seconds. I then ran the program 10 times, recording each output, and created a 10-run average, shown below, to minimise run-to-run variance.

*Recorded FPS Values:* 2748, 2794, 2643, 2782, 2809, 2648, 2700, 2568, 2382, 2397

*Pre-Performance-Improvements 10-Run Average:* 26471 / 10 = 2647 FPS Average

One of the most glaring performance problems in the project is the textures of the critters. Whenever a critter is collided by the destroyer, it gets destroyed, which unloads its texture. Then, about a second later the critter is respawned, which causes another load of the texture. This means that the GPU has to constantly spend resources to load the same texture over and over again. Scaling the project up with more critters would accentuate this issue.

A computer screen shot of a computer code

Description automatically generated This is the code that respawns the critters after they are destroyed. You can see that a new critter is initialised in it’s place of the array, which causes a new texture load, shown in the Critter::Init() method.

A screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated Games wil typically avoid this performance hit by using a resource manager, which will load textures and other resources for objects to access. Resource managers can be configured to know when to unload resource that are unneeded, and generally cause fewer load calls, which can be very taxing on a program such as the one in the project.

For this assignment, I will implement a HashTable that will act like a simple resource manager. A hash table is a table that contains resources, such as a texture, and allows those resources to be accessible by a key, such as “critter\_texture1”. This is very convenient, because the other options would be to load and unload resources, or, like I’ve done before, simply number each entry in a storage container.

A computer screen with text and symbols

Description automatically generatedAlthough using integers in a container, like a vector, is faster than using a hash table to store data, this is less clear to the reader. Calling an index to a vector doesn’t require the program to calculate a hash of a key, and is hence slightly faster. But, hash tables are clearer because they use a descriptive key, making them more readable.

This particular hash table implementation can lead to hash collisions, as there are no protocols in place if two or more different keys create the same hashed index. However, in this project this is fine as only two different textures are being used at any time.

To add a hash table, we will need to change how critters initialise their textures. The first problem is that the critters take in a texture name, and then feed that into a function which finds and loads the texture. Instead, we turn the critter’s m\_texture into a pointer, and feed it an address to a texture stored in our hashtable with a key.

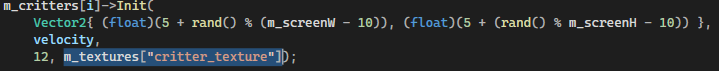
A computer screen with text on it

Description automatically generatedWe initialise our hash table with a Texture2D pointer from the raylib-cpp.hpp header.

Next, we initialize pointers to loaded textures in the Game constructor using the “new” keyword. In this example, I’ve explicitly cast the LoadTexture functions as Texture2D variables so the program would treat them the way we want. I’ve added a small debugger which will close the window with a message if one of the textures fail to load.

A screen shot of a computer

Description automatically generatedAfter this change, we make the Critter initialise take in a Texture2D pointer, and change it’s m\_texture to a pointer type.

Now, using the key, we can pass the hash table into any initialise call for critters using the keyword.

With the performance improvements implemented, we can do the same test as before, and compare the results:

*Recorded FPS Values:* 3019, 2925, 2940, 3049, 3220, 3203, 3189, 3173, 3238, 3203

*Post-Performance-Improvements 10-Run Average:* 31159 / 10 = 3116 FPS Average

Performance Increase:

The performance increase we’re seeing here is around 15%, which is outside margin of error. This means that the implementation of the Hash Table had a calculable improvement to the game’s performance, making it much more optimised. This worked because:

1. Less GPU VRAM is being used at any time, because only one sprite is being loaded that the critters use, as opposed to each individual critter loading it’s own sprite.
2. Less load calls are happening. The critters are not loading and unloading their own textures, but instead are just accessing the texture that has already been loaded by the Hash Table. This is therefore less demanding on the GPU, because it has to load and unload textures significantly less. It is also less demanding on the CPU, as it has to call less complex functions.
3. This is also much more scalable the previous iteration of the project, meaing this project can handle many more critters without as much of a performance loss.

**Class Diagram**

This project, is not really a game, and is not very complex. Hence, the class diagram for the project only contains 3 classes.

A screenshot of a computer

Description automatically generated

Created with Draw.io