# Introduction

In this design brief I will detail several **flaws** I’ve located in the “CDDS\_Optimise” project that could be **optimised**.

The “CDDS\_Optimise” program uses the “raylib” library to display a window, load images, and draw sprites to the window. By default, the program displays 51 critter objects, 50 of them are regular" critters while one is the “destroyer” which appears and functions separately from the rest of the critters.

## Functionality

When the program starts it initializes the window, seeds the random number generator, and then creates an array of 1000 critters. This array is then looped through to initialize the first 50 critters in that array at a random position on the screen and with a random velocity. The amount that it will loop through the array is defined by a constant variable. Then a separate critter variable is created to be reserved for the destroyer, which is then also initialized with a random position and a random velocity.

The critter objects are its own class that holds the position, velocity, radius, and texture of the object along with a few flags. It also contains methods for initializing and destroying the object, as well as updating, drawing, and returning the member variables.  
The initialization method sets the member variables to the parameters passed into the method, loads the texture, and sets the “is loaded” flag. While the destroy method unloads the texture and resets that flag.

During the main game loop, the critters will scatter randomly around the window and can collide with one another. On each frame, each critter will check for a collision against every other critter using a simple circle-to-circle collision check. It will also check for collisions against the window bounds on each frame. If a critter collides with the destroyer, the critter is “destroyed” which as stated before resets the “is loaded” flag. Using a timer, on every second a critter that is “destroyed”, or is flagged as unloaded, will be respawned at the same position as the destroyer and is given a velocity opposite to the destroyers.

# Flaws

## Resource Management

One of the most noticeable **flaws** in this program as evident by the console window that opens along with the game window, is that the *critter’s texture is loaded for each and every critter that is initialized* meaning that there are many unnecessary duplicates of the texture being constantly unloaded and loaded as the critter is destroyed and respawns. This can be easily **optimised** by *loading the texture once outside of the critter object* and then passing the texture to each critter as a pointer or reference on initialization, then unloading the texture once the program exits.

## Memory Management

The next **flaw** I noticed is how the critters are being created. 1000 of them are allocated to the stack within an array but only 50 of them are ever being used. As the stack only makes up small portion of memory, this can be an issue especially in projects with a lot more entities and with objects that take up a lot more data, but for this project it may be more of a minor issue that it is stack allocated.

The simplest **optimisation** I found was to use a **dynamic array** and use the critter count constant to create an array with only the number of critters that we actually want and then delete the array once the program exits. The dynamic array is allocated to the heap rather than the stack, which is a larger portion of memory, though it can be less efficient. While this optimisation works well in this context with a fixed number of entities, in other projects which may have a more dynamic number of entities that may be created and removed from a scene it is a likely more efficient **optimisation** to use an **object pool** instead as to avoid constantly allocating and deallocating objects as that can also lead to memory fragmentation.

## Collision Detection

The program uses a circle-to-circle collision detection algorithm which isn’t a flaw as it’s a very efficient method of collision detection, instead one of the **flaws** in this program comes from it *checking for collisions on each object for every object within each frame*, this can become costly especially with an extreme number of entities. A possible **optimisation** for this may be to use **spatial partitioning** so that the program only has to check for collisions only on objects that are nearby.