Code Design and Data Structures 1: Design game optimisations

By Jevan Soh

# Changes to Implement

* Fix the drawing of critter textures so the centre of the texture sits at the critter's position vector.
* Preload textures into a texture map where each texture has a specified index that other classes can use to access preloaded textures.
* Implement dynamic array class that reallocates memory by doubling the buffer when the count exceeds its buffer and use this to replace the current critter buffer of 1000.
* Implement sorted array object pool for critters. Object pool should have all the active critters at the beginning and all inactive critters at the end. Then remove the m\_loaded variable and the isDead() method as they are no longer needed.
* Remove dead critters from collision checks using new sorted array object pool.
* Implement a prototype class for the critter class to inherit from, then make a new destroyer class that inherits from the same base class. The critter and destroyer class will have their respective radius sizes and texture IDs built in.
* Implement spatial partitioning using a hash table, also known as spatial hashing. This should tremendously speed up the collision checking process as we'll only have to check against other critters in the same and adjacent grid cells. Our spatial hashing system should use radix sort to sort the hash list.

# Further details

## Texture Preloading

Preloading each unique texture into a texture map during the initialisation phase will reduce unnecessary load on the GPU as we can simply reuse the same texture instead of repeatedly requesting the GPU to load and unload the textures whenever a critter spawns or dies. This will technically reduce the task of loading textures from O(n) to O(1). To implement this, first create a texture map class that will store an array of the loaded textures. Then modify the Critter class to store a pointer to the texture map instead of loading and unloading the texture upon constructing or destructing a Critter object.

## Dynamic Array

Creating a dynamic array class which doubles its buffer size whenever the count exceeds the buffer, will mean we won't have to explicitly declare how much memory we want to use when making an array of objects, and at the same time we won't be repeatedly allocating every time we change the array count. The dynamic array class will need three private variables for its count, capacity and a pointer variable to the start of the array. Then add an append method which will check if incrementing the count will cause it to go over the capacity, in which case, the dynamic array should double its capacity and reallocate its data to a new memory block that matches the doubled capacity.

## Object Pool

Implementing a sorted array object pool to store all the critters will allow us to immediately cycle through all the alive critters or cycle through all the dead critters as we wish instead of having to loop through all critters and checking if they're alive or dead. For example, if we had 99 alive and one dead critter, to find and respawn said dead critter, we would have to loop through all critters and check if they're alive or dead, which in the worst case scenario would yield a time complexity of O(n). Whereas with the sorted array object pool, we can quickly find the first dead critter within the array and respawn it with a time complexity of O(1). The ObjectPool class can inherit from the dynamic array class we create and with the addition of its own active count variable which should tell us both how many critters are active and at what index in the object pool the active critter segment ends and the dead critter segment begins. The ObjectPool class should have two methods for loading and unloading objects. When loading an object, the active count should increment. When unloading an object, the unloaded object should swap it's position with the object at the end of the active segment and the active count should decrease by one.

After we've implemented the object pool, we should use it to avoid checking for collisions against unloaded critters. This will not only be less computationally expensive, but will also be more accurate as there will no longer exist the possibility of an alive critter colliding with a dead critter.

## GameObject Prototype Class

For future development it may be useful if we had a prototype class from which the Critter and our new Destroyer class inherited from. To do this we will create a prototype class called GameObject and move the majority of the functionality currently stored in the Critter class to this new GameObject class. Then we will create a new Destroyer class and have it and the Critter class inherit from the GameObject class. Then for both classes we can have their radius and texture ID inbuilt into their class, so we won't have to pass in the same radius and texture ID parameters for every single critter or destroyer we make.

## Spatial Hashing

Currently for collisions we are checking every critter against every other critter which leads to an O(n^2) time complexity that will grow quadratically for each additional critter. Implementing a spatial hashing system will help mitigate this issue as we'll only have to check collisions of nearby critters. There is quite a lot of detail that goes into creating a spatial hashing system, but to summarise we will first subdivide the space into a grid of cells, where each cell has a unique hash. Then every tick we will generate a hash list of critter IDs and the corresponding cell hashes they're located in, use radix sort to sort the array based on cell hash, and generate a lookup array that matches each cell hash to its corresponding start and end index within the hash list array. We want to use radix sort here because it has a time complexity of O(d(n+b)) which under our specific conditions (d = 2, b = 16) will be faster than quick sort's average time complexity of O(nlog(n)). After sorting the hash list and generating a lookup array, we can then quickly find all the critter IDs within any particular cell by simply inputting the cell hash into our lookup array to get the starting and ending indices, and then cycle through the hash list using those indices. After testing the implemented spatial hashing system, we could process collisions of 200 critters at a faster framerate than the old collision system with 100 critters.

# ObjectPool UML Diagram

A screen shot of a computer

Description automatically generated