Critter Optimisations

Problem: every critter is checking collision with every other critter

The part of the critter’s code that is primarily responsible for dragging down the potential frame rate of the program is that every critter object is constantly checking for collision against every other critter object, as well as constantly checking for collision against the destroyer object without taking into consideration either position. This can pass by unnoticed when working with a low number of game objects but as soon as the number begins to be significant then the number of collisions being checked begins to pile up. With the way that collisions are checked at the moment they are using O(n^n) notation which is very inefficient as the number of game objects begins to grow.

Solution: Spatial hashing, reduce number of critters it is checking collisions with.

A paper written by Erin Hastings, Jaruwan Mesit and Ratan Guha from the College of Engineering and Computer Science, University of Central Florida on the topic of optimising large scale real time simulations with the use of spatial hashing [1], goes into a large amount of detail on how collision detection can be optimised by using spatial hashing to reduce the number of checks done per frame.

This process is accomplished by dividing the space into a grid and assigning the different objects to a value depending on their position. From there the objects collisions will only be checked with other objects containing the same value.

Problem: checks every critter against screen bounds

On the topic of using hashing, another part of the project that is dragging down the frame count is every critter checking if their position is past the borders of the screen and if so, flip the velocity based on which border they are past. This check is done every frame on every critter as well as the destroyer and can lead to serious performance issues.

Solution: Only check the critters in the border area

As with the collision optimisation, this area of code can be improved by using spatial hashing to get every critter’s hash value and only run code depending on what area they are in. In this case the check to see if a critter is past the border of the screen would only be done on critters that are in the bordering buckets of the application.

Problem: Having to load the texture every time a new object is created

Every time a new creature is created the program needs to load in the texture for it as well as set all the default values and any other code that runs at its initial load up. This can slow down the program every time it needs to spawn one or more game objects. The program fetches the sprite from its directory every time and can be a waste resources.

Solution: Object pooling

Instead of spawning/deleting a critter and loading/unloading all of its assets whenever we need a new critter or need to delete an existing critter, we can load the required amount of game objects when the program is launched by pooling them together and then removing them from the scene if they are “destroyed” and placing them back in the scene when a new one is needed to be “spawned”.

This is done by allocating a set amount of memory equal to the maximum amount of game objects we expect to have in our scene at any one time, when the program is first loaded. Then setting all of the objects in the pool to inactive until they are needed. Objects that are inactive will not execute any code such as being drawn on the screen or checked for collisions. As an object is “spawned” into the scene they will be flagged as active and begin executing their code without needing to be actually loaded into the scene.

**References:**

[1] <https://www.cs.ucf.edu/~jmesit/publications/scsc%202005.pdf>