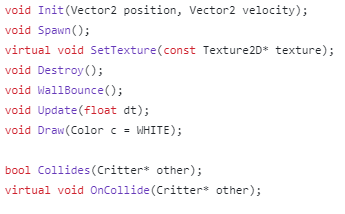
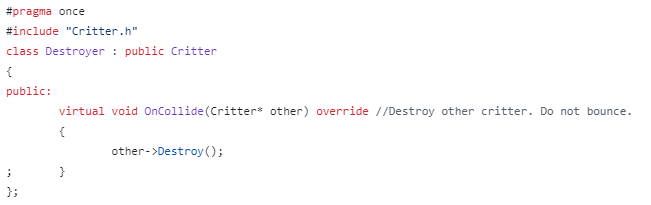
Prototype Design Pattern  
Functionality: The destroyer is updated along with other critters and is treated the same except during collision checks, where the destroyer does not bounce, instead destroying the critter it collides with.

Reasoning: In the previous project there was a lot of repeated code to do the same thing for the destroyer as critters. This way, everything is in methods that both the Destroyer and Critter class use (since Destroyer inherits from Critter), except for collision which the Destroyer overrides.

Critter.h:



Destroyer.h:

Expected Benefits: Improved readability.

Object Pool

Functionality: Neither critters nor textures are deleted when the destroyer “destroys” a critter. For critters, the m\_isLoaded flag is set to false, meaning they do not draw, do not update, and are not added to the Map for collision checks. The critter keeps its texture when this happens. Moreover, there are 2 textures shared between all critters using pointers. Even if the critter itself did unload during gameplay, the textures wouldn’t need to.

Reasoning: There’s no point having more textures than there are images, let alone loading and unloading them during runtime and potentially causing memory fragmentation. Likewise, even though the critters array is now a pointer array (to make the Destroyer’s polymorphism work), deleting critters when they are “destroyed” also means potentially causing memory fragmentation, as well as whatever overhead comes with creating a new object every second.

Critter.cpp:



Critter.cpp:

A black text on a white background

Description automatically generated

main.cpp:

A screenshot of a computer

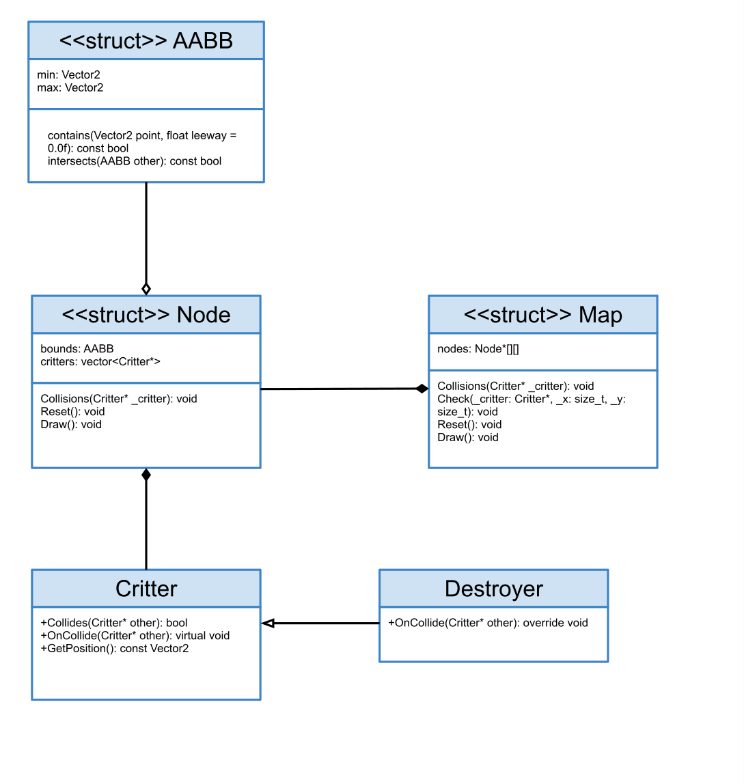
Description automatically generated

Expected Benefits: Improved runtime performance, less memory fragmentation, less GPU memory usage (GPU memory is where raylib says the textures are stored when loaded).

Grid Spatial Partitioning

Functionality:

The screen is divided into a 2D array of nodes (contained in the Map class). It is currently 12x12, but the rows/columns are const values to make them easily modifiable.



At the start of every frame, the map is emptied. During the loop, any living critters will:

* Check what node they should be in based on their position.
* Check for collisions with critters in that node and all neighboring nodes.
* Add themselves to the node they should be in.

This means some collision checks won’t happen from both sides, as the first critter will check for collisions when nothing has been added to the map, and the last critter will check for collisions when all other critters are in the map.

However, since collisions on one side cause both critters to bounce, this doesn’t have noticeable effects. If anything, this helps us. By making the destroyer the last item in the array of critters, we can make sure critters don’t check for collisions with the destroyer, but the destroyer checks for collisions with the critters (and runs its own overridden OnCollide method). That way, the destroyer won’t bounce off critters.

Reasoning: Since checking every critter against every other is normally O(n2), anything we can do to reduce that is important.

Expected Benefits: Worst case scenario for collision checks is still O(n2), but that’s for if every critter is in the same node. Typically, we should expect less operations because critters are only checking for others that were added earlier in the same node/adjacent nodes.

Merged Loops

Functionality: In the original program, not including initialization/deinitialization, there were 3 instances of this loop:

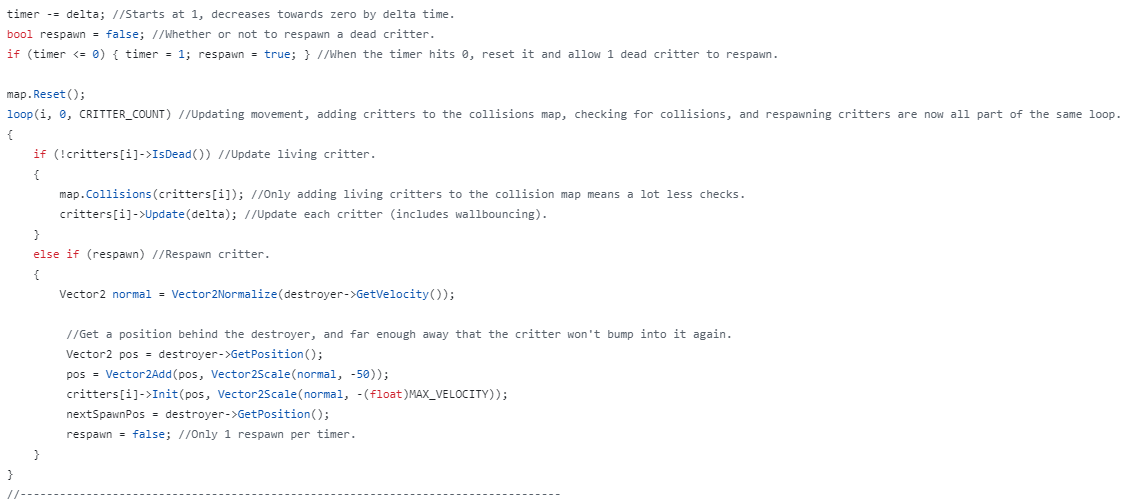
“for (int i = 0; i < CRITTER\_COUNT; i++)”

The first loop updated the positions of each critter, made them bounce off walls, and destroyed them if they were in proximity of the destroyer. Worst case scenario is O(n) with n being the critter count.

The second loop checked every critter against every other critter (skipping ones that have already bounced off another critter) to check for collisions. Worst case scenario is O(n2).

The third loop only ran when the timer stopped, and it iterated through every critter until it found a dead one to respawn. Worst case scenario is O(n), but this is only once every second.

In the below example, critters update, collide, and respawn all in the same loop.



Reasoning: The original 3 loops collectively would be O(n2+n) most of the time and O(n2+2n) when on the frame the respawn timer runs out. If everything was in one loop, the worst case scenario would instead be O(n2) regardless of the timer.

Expected Benefits: Merging the 3rd loop is a minor improvement since it only happens once a second, but merging the 1st and 2nd loop should reduce operations by n (the number of critters).