CS355 Final Project

Adam De Broeck

January 4, 2024

The Idea

The core idea behind this project is to discover parallelization techniques to speed up some existing designs in real time simulations — in this case, physics simulations for video games. In order to find an appropriate avenue to parallelize, some previous techniques need to be examined first — most notably, basic spatial partitioning, which for this project will be uniform grid chunking.

The vast majority of the code and implementations in this project were written from scratch, solely for the purpose of researching this design idea. Some portions of the code will have been re-used as utility functions from other projects such as the shader compilation code written back in GAM200 for a custom engine. Like-wise, the libraries used in this project are the OpenGL bindings library, `GLEW`, the window and IO handling library for OpenGL, `GLFW`, the OpenGL-based linear algebra mathematics library, `GLM`, and finally the UI rendering library, `ImGui`.

Because this project and its graphics were written primarily on a Windows machine and setting up the project for other operating systems would take addition and unnecessary time, the graphical application portion of this project is design to be run on Windows only. That being said, the entirety of the code in this library is written with `C++`.

Additionally, there are limitations to the physics and collision calculations used in the project. Extremely simple bounce resolution and circular collision detection were used, as the actual implementation of these methods doesn't affect the spatial partitioning implementations. For the physics, semi-implicit Euler integration was used, as it is typically easy to implement without much effort.

The application controls for this project are as follows:

```
Left Click - Spawn new ball at cursor
Space - Enable/disable vortex physics
Left Shift - Spawn 1000 balls randomly
Control - Enable/disable graphics
Alpha Num Keys 1-4 - Change optimization type
Numpad Keys 1-9 - Change thread count
```

Real-Time Physics Approximations

As previously mentioned, the physics method used for this project was a semi-implicit implementation of Euler's method. To sum the technique up briefly, each discrete time step of the physics engine calculates the acceleration, velocity and subsequently future positions of all objects in the scene. Acceleration and velocity vectors are scaled by the last frame's average time step in order to account for varying framerate. Mathematically this technique is represented as such:

$$\vec{v}_{i+1} = \vec{v_i} + t\vec{a_i}$$
$$\vec{x}_{i+1} = \vec{x_i} + t\vec{v_i}$$

Where t is the current time step size, $\vec{v}_{i+1}$ is the new object velocity, and $\vec{x}_{i+1}$ is the new object position.

With the integration technique out of the way, the physics engine still needs collision detection and resolution. Fortunately, because this project solely uses circles in the physics demonstration, these calculations are relatively easy. In order to determine if two physics bodies (circles) are overlapping, their radii can be compared to their current distance, as each object has a uniform distance to all points on its circumference. If the combined radii of the two circles in question are larger than the distance between the center of both circles, then the circles must be overlapping. Even better, this method works for any size of circle and mismatching sized circles. In code, the implementation used in this project likes like this:

```
void CheckAndResolveCollision(Ball *collider, Ball *other)
{
    // Check for collision
    float distanceBetween = glm::distance(collider->getPosition(),
                                          other->getPosition());
    float combinedRadii = collider->getRadius() + other->getRadius();
    if(distanceBetween > combinedRadii)
        return;

    . . .
```

After collision detection has verified that two circles are indeed overlapping, the circles must be corrected so that they are forced outside of one another. Again, for circles this is a very easy calculation. The overlapping distance can be calculated via `combinedRadii - distanceBetween`. Once this is obtained, either the first circle can move in the opposite direction of the colliding circle (this works because a collision point of a circle-circle collision will always be parallel to the normal of the surface of each circle's circumference) or both circles can move half the distance each. In this implementation, both circles will move half the distance. The rest of the previous function can then be completed via the following resolution code:

```
   . . .

   // If colliding, move the collider out of the other ball
   glm::vec2 direction = other->getPosition() - collider->getPosition();
   if(glm::length(direction) < 0.0001f)
   direction = glm::vec2{0.0f, 1.0f}; // Account for perfect overlap
   direction = glm::normalize(direction);

   glm::vec2 reversePen = -1.0f * direction;
   reversePen *= (combinedRadii - distanceBetween);

   collider->setPosition(collider->getPosition() + (reversePen / 2.0f));
   other->setPosition(other->getPosition() - (reversePen / 2.0f));
}
```

The final piece of the puzzle to have a half decent physics implementation for circles is to have each circle rebound off of other circles and the walls of the space. Typically this rebound amount is a float scalar from `0.0f` and `1.0f` which represents the percent of the ball's velocity that is preserved in the elastic collision. This works perfectly for the walls of the space because they are aligned with the axis of the world — therefore, to calculate a proper rebound, the x or y component of the ball just needs to be reversed and multiplied by the ball's restitution value.

For ball-ball collisions however, the ball must bounce appropriately off the tangent of the collision point for any given ball which can occur at any point on the ball's circumference. It turns out, this isn't too difficult either, however it requires some additional linear algebra.

To get the correct rebound vector, the tangent to the collision point must first be calculated. This can be done by getting an orthogonal vector from the direction vector (which is subsequently the normal vector of the collision point) by using the "switch and flip" technique — switch the x and y components and flip the sign on either of them. In R2, this will always give an orthogonal vector.

At this point, the correct rebound direction can be found by getting the orthogonal projection of the velocity vector of the colliding ball onto the new tangent vector. Once this vector is found, subtracting it from the current velocity twice will give the correct rebound vector for any collision angle in a circle-circle collision.

That's a bit of a mouthful, but it can be all summed up in one image:

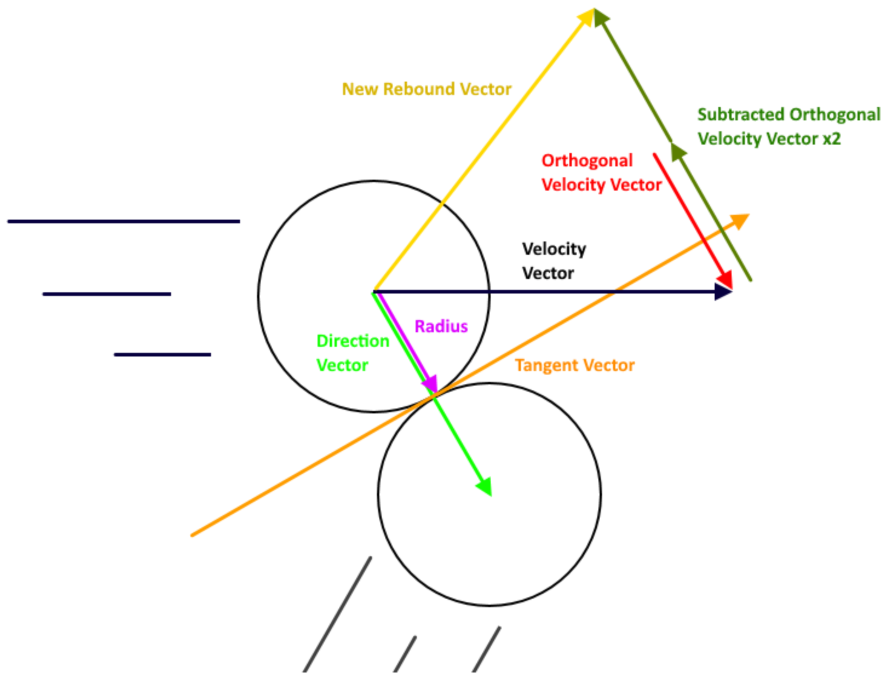


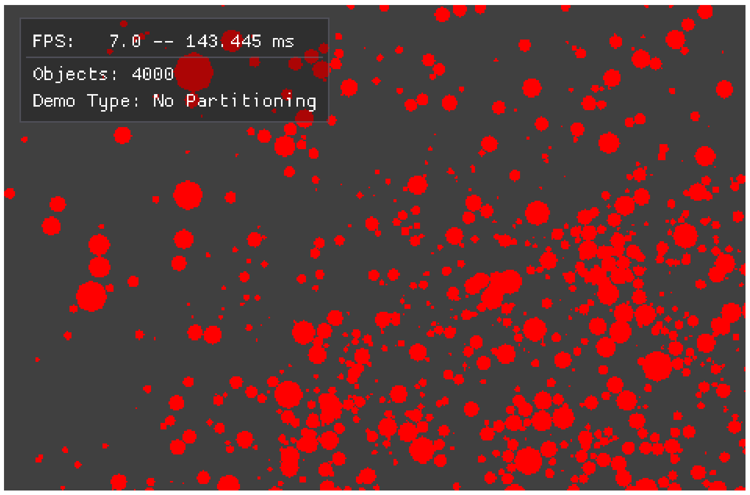
Figure 1: Ball-Ball collision and rebound resolution.

A Naïve Approach to Collisions

A typical first approach to checking collisions in a physics engine is to iterate over all of the physics objects in the scene and integrate their position. After integration, the objects are then subject to collision detection and resolution for other objects. A naïve approach to this is to simply loop over all of the objects again, for each object, and check collisions (ignoring the same object of course). It doesn't take much to immediately notice that the growth pattern of such a method is $O(N^2)$ and does not scale for large numbers of objects at all.

There are a number of ways to improve upon this system. Some involve polling the space in front of an object which is owned by a top level container, others use clever "boxing" methods to only check for collisions that have a possibility to occur. Both of these techniques are spatial in nature, however, for this project the latter will be used. Specifically a system of grids called a chunk map. The purpose of this map is to keep spatially local objects together so that they don't need to check collisions with all other objects. This method doesn't eliminate the $O(N^2)$ growth pattern but instead makes the scaling properties of such a pattern nigh impossible to have occur in practice. Such a method might be referred to as a "Divide and Conquer" method.

Below is an example of an $O(N^2)$ approach to collision detection with no spatial partitioning:



```
FPS:   7.0 -- 143.445 ms
Objects: 4000
Demo Type: No Partitioning
```

Figure 2: Choppy framerate from a bad physics implementation.

The above simulation typically runs at a locked `144fps` and `6.5ms` frame time on the testing hardware (more details on that later). As can be seen here with the naïve implementation, frame times skyrocket to more than 20x that of the original application.

Spatial Partitioning Implementation

As briefly mentioned before, the solution to this problem can be found in spatial partitioning. The implementation details for the partitioning method used in this project are like so: At the beginning of the application, create a hash map that takes coordinates as a key and stores a hash set of physics objects as the key. Whenever an object is spawned, the object registers its position in the hash map, accurately named `chunkMap`. Each frame, the chunk map is cleared and the positions of all objects in the scene are re-cached into the chunk map. For simplicity sake, the coordinate space of the chunk map is simply the position of the physics object but truncated to a 32-bit integer rather than a float.

Next, when iterating over the physics objects in the scene, instead of iterating over all of the objects, iterate over the chunks available in the map. For every physics object in a given chunk, integrate it and perform collision chunks with all adjacent chunks. This prevents physics objects from needing to calculate possible collisions with objects that are too far away. The size of a given chunk affects how well this technique will work. Too large of chunks will fail to localize enough collisions, and too small of chunks will require magnitudes more overhead in adjacent chunk calculations.

There is a slight problem with this technique which is quite similar to another known physics simulation phenomenon. Objects that are moving very very quickly may pass directly over chunks or objects themselves. They could even pass outside of the adjacent chunks used for collision detection.

In this project, this problem was ignored as objects typically didn't move that fast nor was it a concern for later parallelization efforts (obligatory, "leave this exercise to the reader"). Just by introducing a chunk map into the mix and using basic spatial partitioning techniques the previous example with 4k objects in the scene goes from a whopping `7fps` and a `143ms` frame time to `116fps` and an `8.6ms` frame time. That's almost back to the original frame time that uses no collisions at all!

Below is an example of that chunking system in action:

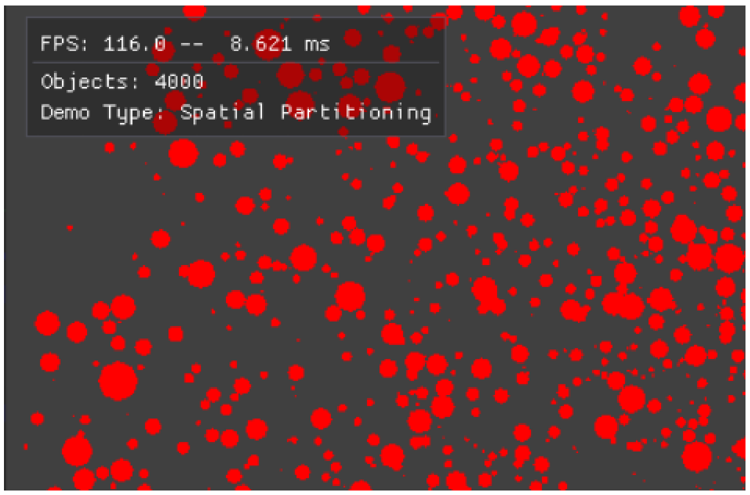


Figure 3: Stable framerate using chunked spatial partitioning.

Multi-Thread The Chunking

Everything so far has just been an appetizer for the main course. Again, the real reason for this project was to find avenues for parallelization. Now that the chunking system is in place, there's a unique property that the spatially local collisions can now guarantee. Because the chunk map will never do calculations for physics objects outside of adjacent chunks, it's practically free for optimization via multi-threading. All this needs is a pattern on how to "hand-over-hand" the data and the rest will fall into place.

Given the grid-like nature of the chunk map, it's tempting to use a checkerboard pattern as a start. For the most part this works pretty well if you ignore the corners and it would allow for 2 passes over the entire simulation space. Each pass would queue up jobs to be put into a bucket which is then accessed by a thread pool and processed. Because of this, it is also simple to use any amount of threads desired.

A theoretical/graphical implementation of the checkerboard pattern might look like this; where yellow is the currently operating-on chunk by a thread, blue are other possible operating threads green is a valid chunk that no thread is writing to and read is a thread that is being written to:

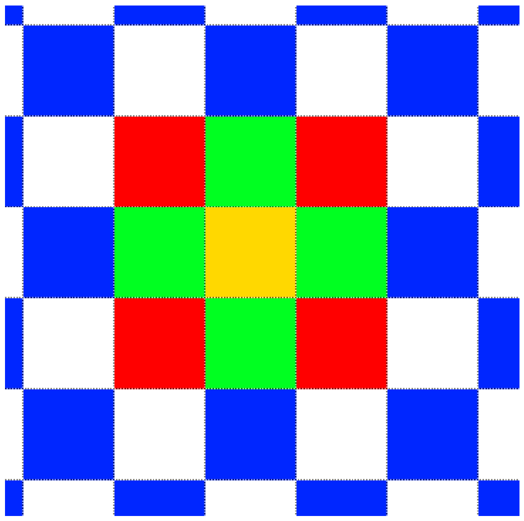


Figure 4: Checkerboard fails to get concurrent non-overlapping access to each chunk in one pass.

To fix the above problem, the space between working tiles can be increased so that there is always a space in-between, even on diagonals. This solves the concurrent writing issue, while increasing the total number of passes to 4. The outside loop can then traversal all chunks in the space by using a `Z` patterns.

This is what that might look like:

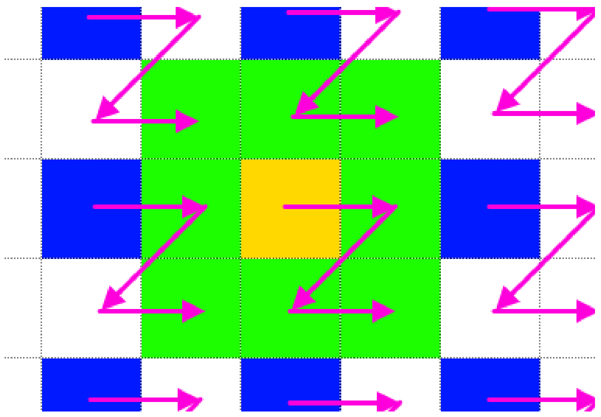


Figure 5: Fixed diagonal chunk overlap, passes increased to 4.

Unfortunately, this works if collisions detection and resolution only acts on the colliding ball which is owned by the center chunk. However, a decent number of collision resolution techniques prefer to act on **both** colliding objects as it makes the physics more stable — this project also acts on both objects. Because of this possibility, the thread that acts on the center chunk **must** have write access to adjacent chunks in order to store the second half of the collision data back into the other colliding object. This can be fixed by doing essentially the same thing as the previous fix — moving the chunks further apart.

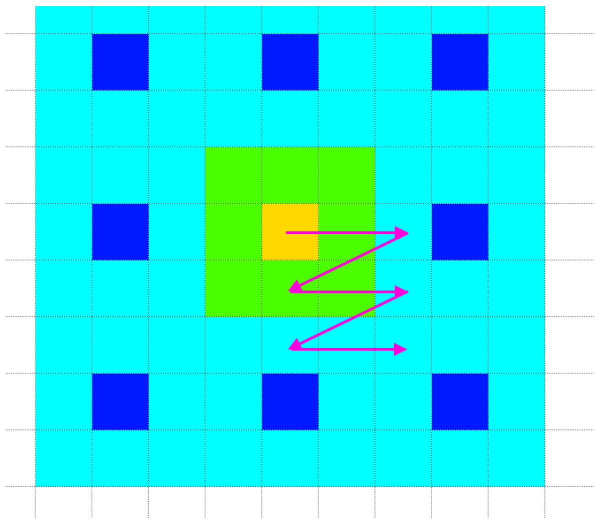


Figure 6: Fixed adjacency write issues again, passes increased to 9.

This adds a minor amount of overhead to the entire process. However, the amount added is near negligible in comparison to how expensive the collision detection and resolution code is — therefore, it's an acceptable tradeoff.

The last part of the multi-threading technique is to write the thread pool implementation. In order to easily get all of the threads to run at the appropriate time to process a singular batch, a barrier was used. Specifically a `std::barrier` which was added to the `C++ STL` in standard version 20. This is nice, because it means a new barrier implementation wasn't a necessity to write for this project.

The code that batches and processes all of the adjacent chunks into the thread safe queue (uses a mutex) is as follows:

```
if(physicsType == PhysicsType::THREADED_CHUNKED_PARTITIONING)
{
    for(int i = 0; i < 9; ++i)
    {
        if(numThreads == 0) break;

        int chunkXOffset = i \% 3;
        int chunkYOffset = i / 3;

        // Create work orders for each chunk
        for(int chunkX = -10; chunkX <= 10; chunkX += 3)
        {
            for(int chunkY = -10; chunkY <= 10; chunkY += 3)
            {
                Util::Coordinate coord(chunkX + chunkXOffset,
                                       chunkY + chunkYOffset);
                if(chunkMap.find(coord) == chunkMap.end()) continue;
                if(chunkMap.at(coord).empty()) continue;

                // Generate work that needs to be done
                WorkOrder order;

                . . .

                // Add to work queue
                // (technically this doesn't need to be locked)
                {
                    std::lock_guard<std::mutex> lock(workQueueLock);
                    workQueue.push(order);
                }
            }
        }

        // Start thread pools
        rendezvous->arrive_and_wait();

        // Wait for thread pools to finish
        rendezvous->arrive_and_wait();
    }
}
```

(It's possible to use the main thread to do work as well but there wouldn't be any addition gains as it would be no different than the main thread sleeping while another thread operates — effectively it'd just be like adding another thread)

Performance Results

Initial results from the spatial partitioning implementation over the naïve implementation were already very good, however, the multi-threaded performance increase was staggering. After further analyzing the data, like most multi-threading approaches, the gains from adding more threads typically tapers off with diminishing returns.

Surprisingly the multi-threaded approach was so good that it **almost** kept up with the baseline `No Collisions` demo (full framerate) up until 8,000 objects, which is typically when the graphics pipeline starts to slow down from driver overhead if a user doesn't switch to using instanced draw calls (something that was outside the scope of this project). Because of this, a keybind to disable the renderer was added in order to get more accurate results for larger object count tests.

Below is a table of the average frame timings of each method with a given object count:

Object Count	No Collisions	Naïve Collisions	Spatial Partitioning	Multi-Thread (2)	Multi-Thread (4)	Multi-Thread (6)	Multi-Thread (8)
1,000	6.9ms	12.2ms	6.9ms	6.9ms	6.9ms	6.9ms	6.9ms
2,000	6.9ms	45.5ms	6.9ms	6.9ms	6.9ms	6.9ms	6.9ms
4,000	6.9ms	194.4ms	8.7ms	6.9ms	6.9ms	6.9ms	6.9ms
8,000	6.9ms	864.0ms	32.4ms	17.6ms	11.2ms	9.8ms	9.2ms
16,000	6.9ms	3,770.0ms	143.3ms	91.5ms	59.6ms	49.2ms	49.8ms
32,000	10.2ms	15,820.0ms	942.0ms	191.0ms	172.4ms	169.5ms	169.2ms

When graphed, the changes in speed become even more apparent:

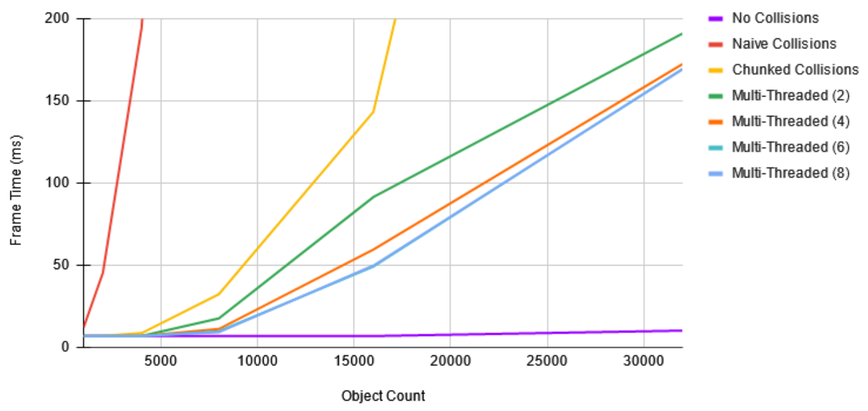


Figure 7: Graph of each method's timings over the number of objects tested.

Additionally, here is a graph of the scalar magnitude increases in speed compared to the naïve implementation as the implemented method gets better for each object count test.

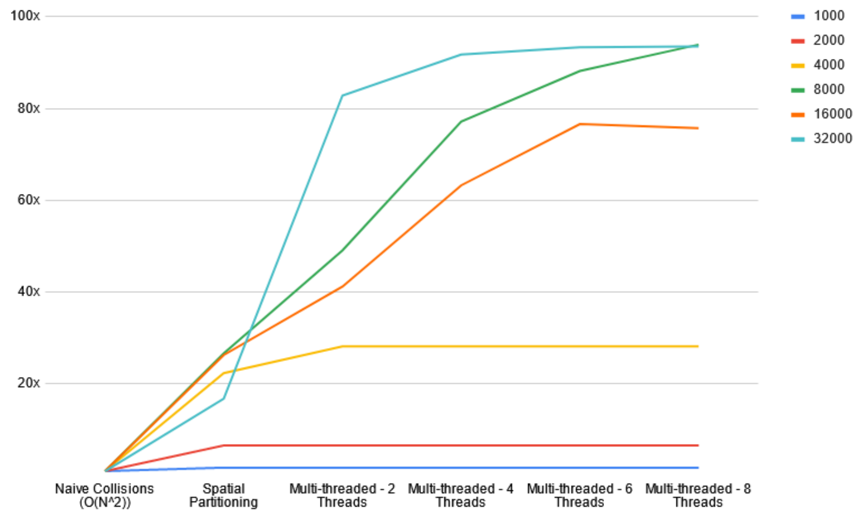


Figure 8: Object count performance gains over baseline as methods change.

Possible Extensions

This project only served as a theoretical baseline to how spatial partitioning can be parallelized. There are many other techniques that might work even better than grid based chunk.

When a couple ideas come to mind, the first which might be a significant challenge but could pose a very interesting result is using dirty quadtrees instead of chunking. Typically these types of structures are much more efficient than having constant sized grids. As long as a user could effectively determine if a quad tree was overlapping, the same principles of parallelization can apply. It's possible to use Axis Aligned Bounding Box detection for such an implementation.

Another possible improvement would be the further optimization of code and/or data structures used in the project. The only major optimization that was thought of before starting the project was to use hash maps for the chunk structure as hashing of coordinate pairs is extremely quick. There may be other data structures or algorithms that could speed up the rest of the implementation. One thought that comes to mind is not re-populating the chunk map every frame, but rather re-registering objects to the chunk map only if they cross chunk boundaries.

Finally, this implementation chooses to completely avoid what happens if an object is moving too quickly to be considered in either its parent chunk or adjacent chunks. This means that objects that move too fast tend to completely skip the collision detection phase which is not desirable. A possible solution to this would be to do a pre-pass to get the fastest moving object and divide the time step into multiple passes such that each pass cannot physically allow the fastest moving object to cross a whole chunk in one frame.