Silas Raphaelidis

Delving Into Raymarching, Compute Shaders, and Post Processing

For one of my college courses, I am currently working on an immersive experience. The premise is that the player has a heart rate monitor attached to them, and is placed in an infinitely generated dark cave with nothing but a small light source. There will be creepy noises and other ambiance, but nothing bad will happen until the player's heart rate begins to rise. Once that happens, things start getting weird. Monsters will start to creep toward the player, getting faster as their heart rate goes up. The monsters continue to speed up until they either catch the player, or the player calms down enough to lower their heart rate.

For this project to work, I needed quite a few interesting systems I would have to build.

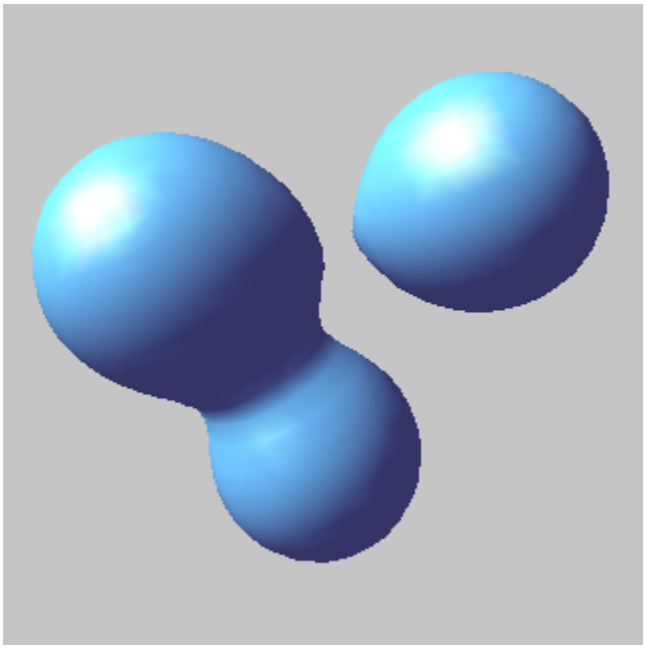
- The Infinite Dark Cave
    - I already have an endless dungeon generator I made for another project that I should be able to repurpose for this one.
- Getting the heart rate monitor hooked up to Unity:
    - I bought a Coospo HW706 Heart rate monitor and found an API I can use at https://pulsoid.net/ . While this seems to be primarily used by streamers, I should be able to hook it up with Unity using a webhook, though I haven't gotten the chance to try it yet.

- The Monsters
    - My plan for the monsters is to use a boids flocking algorithm. As the player's heart rate rose the boids would move faster toward the player, only backing off when looked at. I hope that forcing the player to look for the monsters to keep them at bay would add a sense of paranoia.
    - Luckily I already made a boids algorithm for another class project, and repurposing it for this one was relatively easy.

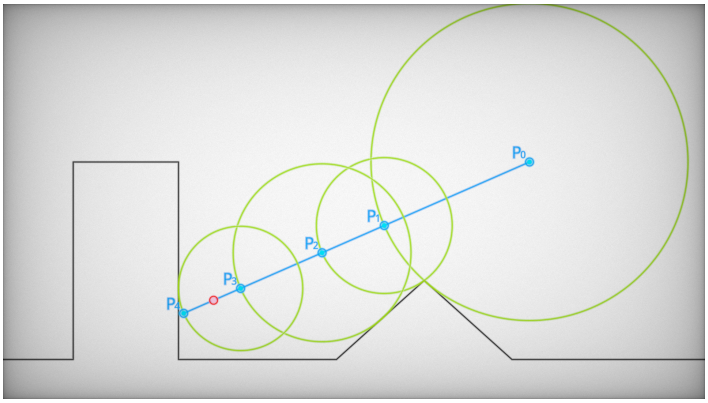
- How to make the monsters scary enough to raise people's heart rate:
    - This is the core issue I'm tackling in this write-up.
    - Boids by themselves aren't very scary, so I wanted to create some kind of trippy shader effect that made it less obvious what was happening, keeping some sense of mystery around the monsters.

-Talk about my foray into metaballs, how they work, and how mine didn't.
-Talk about my metaballs approach and why it didn't work

My first idea was to use the Metaballs effect for the monsters. This effect characteristically has shapes that morph and meld into each other as they get close. The image to the right is what metaballs should look like. My idea was to attach a metaballs effect to my boids system to have a flock of spheres look like an amorphous blobby sludge flying toward the player.



Metaballs is an effect that is hard to achieve through a normal rendering process. Instead, we use a technique called ray-marching to render them. In order to use raymarching, we first have to define all of our objects using the effect as SDFs (Signed Distance Function). A SDF is a function that given a point, returns a positive distance if the point is outside the object, returns a negative distance if the point is inside the shape, and returns 0 if the point is on the edge of the shape.



Raymarching Visualized Below

Using SDFs we can now start raymarching. Given a starting point and direction, we loop through the SDFs and find the minimum distance from the SDF to the starting point defined as D. Next we move along our given direction by distance D to define a new point. We repeat this process until we reach a point where D is very close to 0. When this happens we assume we are on the edge of an SDF.

To use Raymarching as a rendering technique we need to raymarch out from each pixel on screen. This can be done using a fragment shader in Unity. Now we have a scene that renders circles on the screen where the raymarching hits the SDF. To make the Metaballs effect with this all we have to do is replace our minimum distance to an SDF with a Smooth Minimum function. This means if a pixel is marching between SDFs, if it finds the Smooth Min to meet its threshold then it will render.

The effect works, but there are some major issues. Performance is terrible, I'm only getting around 20fps with 5 boids on screen, and adding more plummets the framerate even further. This is happening because, at each step along every ray, I'm looping over every SDF to find a smooth minimum. Most pixels in the scene don't hit an SDF, so I have a lot of extra processing that isn't resulting in anything. There are ways to optimize this to run more efficiently that I could have implemented, but there was a bigger problem at hand. I didn't like how the effect looked. The boids didn't have any scare factor to them, which defeated the whole purpose of using the effect. So instead of improving my metaballs, I decided to pivot to something else.

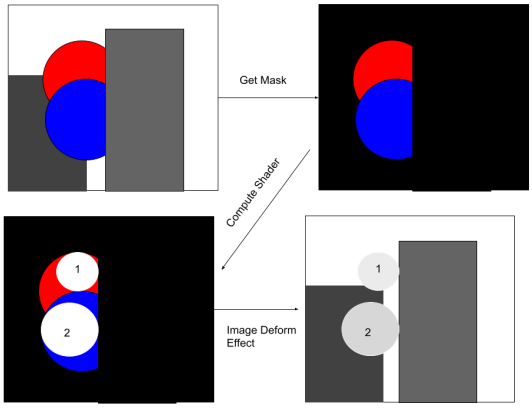
Keeping in mind the problems that arose with my Metaballs approach, I started brainstorming my next approach with two things I wanted to accomplish:

- Performance first. Though the metaballs shader looked good, the frame drops made it entirely unusable, and even if I had done optimizations, the system still likely wouldn't be able to handle as many boids as I wanted.
- Obscure more from the player. Once the player saw the metaballs once or twice, all sense of mystery was lost. A horde of metaball boids is visually indistinct from a mass of sludge. And sludge doesn't make you paranoid.

My first idea was to overlay a circular screen distortion where the boids were visible, making the distortion more intense as they got closer to the player. This would hopefully make it a bit less obvious what was happening to the player when they saw the boids from afar. A slight distortion that runs away when looked at should give the player a moment of "Did I just see something move?", which is exactly what I want.

I also want to mess with the player's sense of depth, so when a boid goes behind another object, and is only partially visible, instead of cutting the distortion effect in half, I want to create the largest full circular distortion effect that can fit within the visible area of the boid. This visual effect would take three separate steps:

1. Use a fragment shader to get a mask of the pixels where boids are visible.
2. Use a compute shader to get a list of the positions and radii of the largest possible circle that can fit within the visible area of each boid.
3. Pass that list of positions and radii to a fragment shader to perform a distortion effect.

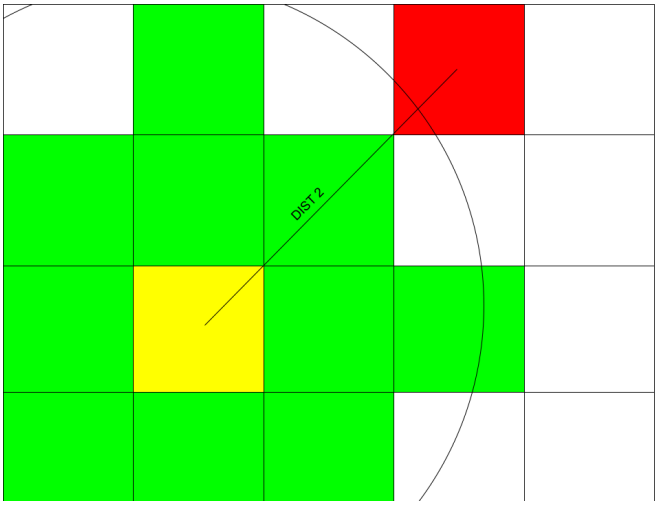


Visualization:
Red and Blue circles represent the boids, and grey circles 1 and 2 represent where the distortion effect will be applied.

The first step I implemented was the distortion image effect. This was by far the simplest of the steps. Given a list of positions in screen space and radii in pixels, Each pixel in the frag shader will check each circle's center position and radius to check if the pixel falls in the range of the circle, if it does, we apply a circular uv distortion, with the strength of the distortion based on how far the pixel is from the center of the circle. The render texture we pass to this shader is from a camera that does not have the boids included in its culling mask, so the distortion applies to the objects behind them.

The second step I implemented was the fragment shader to get a mask of the pixels where the boids were visible. To do this, I used two cameras. Camera 1 can only see the boids in its culling mask, while Camera 2 can see everything but the boids. Then in my fragment shader, we read the depth buffers from each camera and at any pixel where Camera1's depth is closer than Camera 2 we render what Camera 1 sees. Otherwise, we render black.

The last and hardest thing to implement was the Compute Shader. My original approach was to create a thread for each pixel on the screen, and have that thread step out along the cardinal directions, and diagonals, comparing the colors at those pixels to the color at the starting pixel. The thread continues stepping out until it reaches a different color pixel, deeming that the edge of the shape. Note that this is an approximation of the distance to the edge. An approximation was fast and accurate enough for my use case, but I would have found a different algorithm if I needed more accuracy.



Then the amount of steps performed is encoded into the return compute buffer at that pixel's index. This process is visualized in the image to the left. The compute buffer with all the encoded data is then sent back and read on the CPU.

On the CPU, each value on the compute buffer is read, and we compare each distance from the edge to find the pixel with the largest distance from the edge. We deem that value as the center of a circle, and the radius as the distance from the edge. We can then pass those values onto step 3, our post-processing distortion effect.

This approach gave the desired result, but it had massive performance issues. There was a huge bottleneck when we read the data from the GPU back to the CPU.

Reading an entire compute buffer with an element per pixel every frame was too slow. I needed to reduce the amount of data I was sending back to the CPU.

I needed to offload the algorithm on the CPU that finds which pixel has the largest distance from the edge to the GPU so that all the CPU would receive would be a list of Vector3s for each circle with (Circle x-position, Circle y-position, Distance from Edge/ Radius). To do this I would need to compare the distance from edge values on the GPU, which is a problem because the threads can't directly talk to each other or write to global memory without causing data races. I couldn't use group shader memory because the groups couldn't talk to each other and there might be circles that have pixels in different groups.

After digging through lots of forums and documentation, I found the HLSL function InterlockMax(), which compares a value in global memory with a value in the thread, and if it's greater, we replace the global memory value with it. This lets us find the biggest value across each thread and return that back to the CPU. The only caveat is that in Unity's version of HLSL, InterlockMax can only compare 32 bit uint values. We can't compare Vector3s like we need to because we can't define a custom comparison function for interlock max.

To get around this limitation, we need to compress and encode our Vector3 into one uint so we can use InterlockMax. In our uint, we encode the distance from the edge in the leftmost 10 bits, the y position in the next 11 bits, and the x position in the last 11 bits. We use this ordering so when we compare this uint using InterlockMax, it compares using the distance from the edge because those are the most significant bits. In the case there is a tie, it compares using the y value. Note that this approach doesn't work in the case of a 4K monitor, because there aren't enough bits to encode all the values in the worst-case scenario. To deal with this edge case we either need to lose 1 bit of precision for each value, or force the game to run at a lower resolution.

When we use our new algorithm and pass the resulting list of positions and radii back to the CPU the performance improves significantly, from around 20-40 fps with the old version up to 70-90 fps with the new algorithm. This is a great improvement, but we can do better than this.

The biggest bottleneck is still reading data back to the CPU, luckily we can skip this step entirely by sending our output compute buffer directly to our next step, the fragment shader. This means the CPU doesn't have to read the values on the compute buffer at all, and all the data just stays on the GPU. To get this to work, we need to change our post-process distortion effect to take in a list of uints instead of screen

positions and radii. Then it takes the uints it receives and decodes them into Vector3s representing the position in the x and y values, and the radius in the z value. It can then use those values exactly as it did before to achieve the effect.

This optimization skyrocketed the performance all the way up to 210-230fps, and about 90-110 fps when the boids system is active. While the boids system could use some optimization, we are still above 60fps at all times which is great.

There are other optimizations I could do to boost performance even higher.

- My algorithm to step out and find each pixel's distance from an edge could step 2 or 3 pixels at a time, losing a bit of precision, but speeding up the algorithm significantly.
- I could find a different approach to step 1, where I get the mask of visible boid pixels. If I used an approach that only used 1 camera instead of 2 it would be more efficient because having multiple cameras in a scene is expensive.
- I could use techniques to cull boids that are not in view from all calculations

If I return to this project, these improvements would be worth implementing, but for now, I am happy with the result.