Comp 3490 Final Project Report

Devin White, Ben Samsom

Overview

For our project we decided to make a space game where you destroy randomly generated voxel asteroids rendered with the Marching Squares algorithm in a mining ship that fires space rock melting lasers. However, if you think about it abstractly enough, it’s really just a massive claw machine. The ship is the claw, the asteroids are prizes, and the infinite expanse of space is the machine. So we really implemented the best possible extension to A1 and A2 we could think of. All of the files we directly created or modified are in the Project directory. Any modified or referenced code has been cited in comments. Unmodified resources are located in the Resources directory. All references are also listed at the bottom of this document.

The Game

Our game is an asteroid mining simulation where you pilot a mining ship and carve up asteroids in space. You start by selecting a region in space to form an asteroid field from a randomly generated simulation, then watch as the asteroid field generates around you. Your ship is equipped with a number of subsystems, including a mining laser, a distance scanner, and auto turrets. You can navigate through, explore, and mine the asteroid field.

Instructions and Controls:

There are two parts of the project: Sim/Game Mode, and Demo Mode.

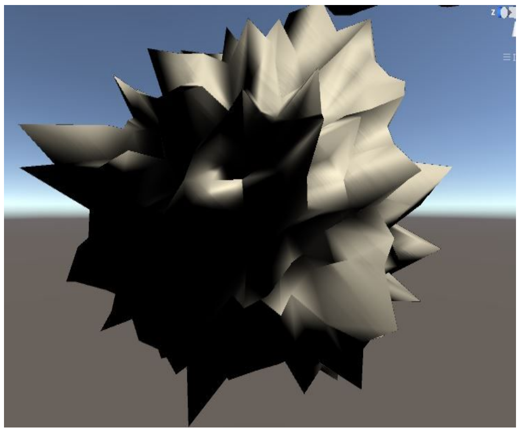
In Sim/Game Mode you first select a region in space for an asteroid field. You will need to capture a volume of points in the particle simulation to determine where your asteroids are generated. You can move the camera with 'WASD', freeze/unfreeze time with 'Space', 'Left-Click' to generate a bounding sphere around the camera, and 'Right-Click' to remove the bounding sphere. Press 'Enter' when you have selected your preferred space to move to the playable part. Press ‘Escape’ to return to the main menu.

Once you have chosen your asteroid field, you control a ship and fly through the generating field modeled after your selection. The asteroids are generated from voxels, meaning they are fully destructible with the use of the ship's mining laser. They will be spawned in over time, as the calculations to generate them are fairly intensive. You can watch the field generate around you, or set out towards already spawned asteroids. One asteroid will always be spawned in front of the ship’s starting location for convenience. The ship uses Unity physics and is controlled by applying forces. It can be controlled using 'WASD' for directional movement, 'Space' and 'Shift' to move up and down respectively, and 'Q' and 'E' to apply roll force. The mouse is used to control directional force. The target in the center of the screen is neutral, so pointing the mouse there will not apply any new force. Moving the mouse anywhere else will apply a force in the corresponding direction. Pressing 'Z' will enable/disable flight assist, which if on attempts to neutralize forces. The ship has two camera viewpoints, Observer (default) and Front, which can be switched between by pressing 'V'. On Observer mode the camera can be zoomed in and out by using the 'Mouse-Scrollwheel'. Holding down 'Left-Alt' in either camera switches the camera to free-look mode. The ship has several subsystems. Pressing '1' fires the mining laser, which will destroy asteroids that it hits. This has a heat slider displayed in the UI. Pressing '2' activates the radar, which pings surrounding objects. Pressing '3' activates turrets, which will shoot at nearby objects. You can return to the Sim Mode main menu at any time by pressing 'Escape'.

In Demo Mode you are presented with a randomly generated asteroid, and can ‘Left-Click’ it to destroy it. You can spawn a new asteroid by pressing ‘Enter’, play/pause rotation by pressing ‘Space’, and return to the main menu by pressing ‘Escape’.

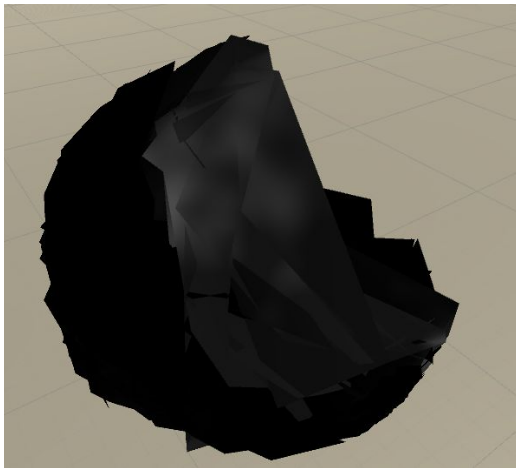
Asteroid Generation

After looking at a few sets of unity assets we realized everyone cheats and uses three of four different asteroid models for all of their asteroids. We didn’t like that and decided to make our own procedurally generated asteroids.



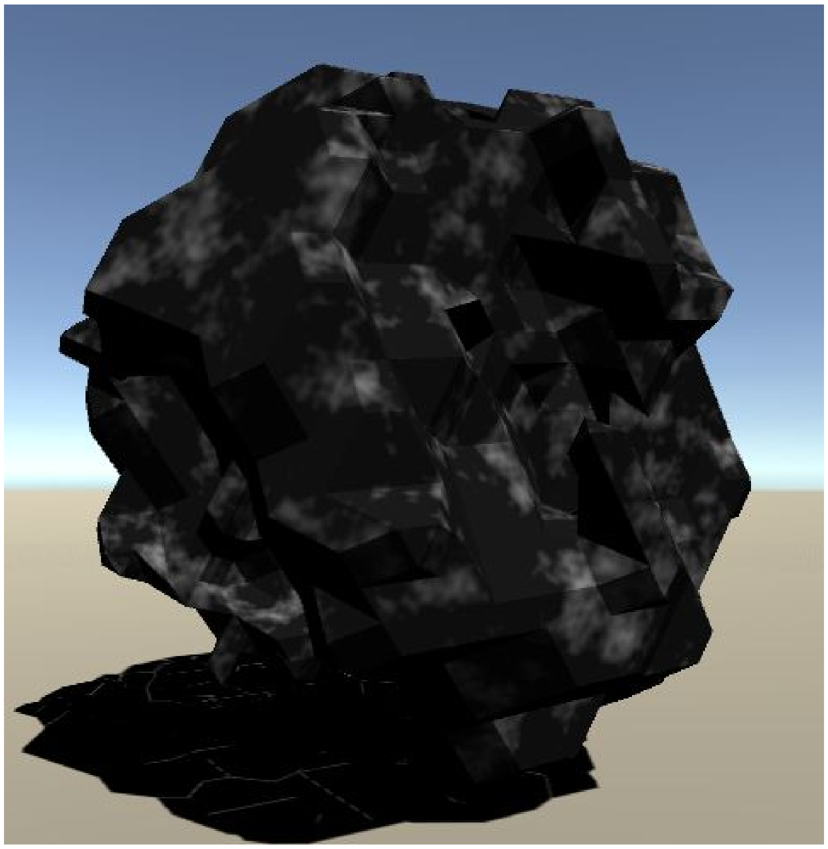
Early on we came up with two ideas of how to do this: generate our own spherical objects and deform them before they are created, or take a sphere and break it. Being typical computer scientists, we decided to take the pre-existing sphere objects and break and deform them to suit our needs. After all who doesn’t like breaking things? Even after deforming the spheres in a random fashion they still looked similar(as shown to the right).

To avoid similar looking asteroids we used random surface deformations and Perlin Noise to make each asteroid a different looking sphere. Once we finished all the deformations and rendering the new mesh we applied a Perlin Noise texture onto it via a custom shader which we will discuss later in the document. (result is shown below)



We created asteroids by starting with a sphere and running an algorithm(WarpVertices) we created on them. WarpVertices does two things. First it takes a set of input vertices and applies a uniform function of random nature over the x, y and z values. Then it takes this new shape that doesn’t have a uniform surface and runs a set of Perlin Noise on it, randomly choosing between increasing the values(adding hills) and decreasing the values(adding gullies). This set of Perlin Noise came included with the Marching Cubes algorithm we found.

Once the shape is nice and deformed we apply the vertices to our mesh and recalculate the bounds. If you render this new shape at this point it looks like a spiky glass ball (above) that has surface shatters. We then use the mesh data to turn the shape into a 3D Voxel array using a Voxelizer algorithm which uses AABB trees. We obtained the algorithm from the same source as the Marching Squares algorithm.



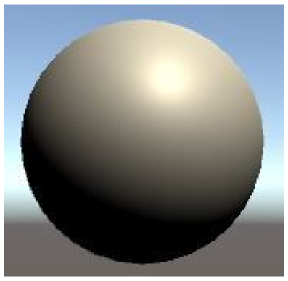
We tried to run Marching Cubes at this point and realized it was cutting off the edges of our new warped shape on all sides. To compensate for this we padded the voxel array before we passed it onto a Marching Cubes algorithm. The Marching Cubes algorithm we found takes a 1D Voxel array as an input, so we wrote a 3D array to 1D array converter. Using this new flattened voxel array we run Marching Cubes and reapply the mesh to the original sphere. Upon rendering we get these amazing sphere objects with lots of deformations that are similar to asteroids. Our new shape was nearly impossible to uv map to as it had such an odd shape, so we modified a soon to be detailed Perlin Noise shader to vary the asteroid surface. (result shown to the right)

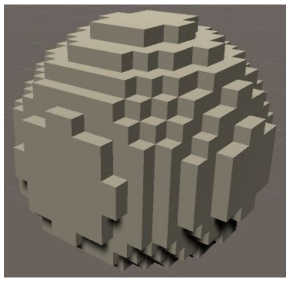
Destructible Asteroids

Once we had created our custom asteroids we had make a way to mine them. To test this we created a new scene that has a single asteroid sitting on flat plane and a basic skybox (as featured above). We used this simplified scene to make asteroid destructible. This scene does a raycast from the mouse to the asteroid. Transforming the click location from screen space to world space and then to the asteroid's object space gave us a hit point on the asteroid. With this location we could round to the nearest voxel and remove it, recalculating the mesh by re-running Marching Cubes and updating the vertices, triangles, normals, and setting the collider mesh to the new mesh we've created. At this point we have this really neat Scene where you can click repeatedly on the asteroid to remove Voxels one at a time.

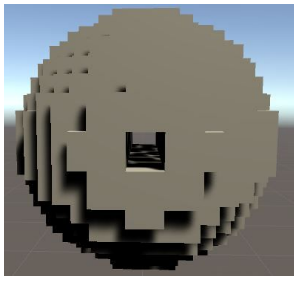
We applied the same effects using a raycast from the ship to the asteroid whenever you shoot the laser, as well as when the turrets fire.

What is a Voxel?





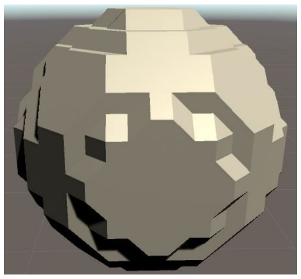
Voxels are a way of representing an object by storing a 3D grid of points. Voxels can then be converted to meshes by converting them to vertices. On the Left we have a normal sphere, on the right we have the same sphere but represented by a simple voxel to mesh algorithm that adds quads around each voxel. While looking at a voxelized object may be less appealing, it allows you to manipulate the mesh way easier. For example these spheres are using a 16 x 16 x 16 array of voxels. Let's say you wanted to bore a hole in the center of this sphere, is not obvious how you might do that with the sphere on the left. However, the sphere on the right is simple. Because it's using a Length, Width, and Depth of 16, you know that half of its Length and Width is the midpoint, to bore a hole you would just loop through the 3D array at Length/2, Width/2, over its entire depth setting these voxels to zero.

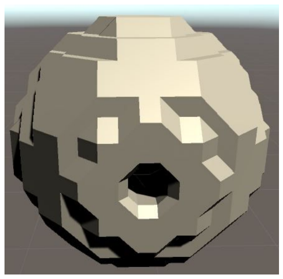


```
for(int i = 0; i < size; i++){
        m_voxelizer.Voxels[(size / 2)     , (size / 2)     , i] = 0;
        m_voxelizer.Voxels[(size / 2)     , (size / 2) - 1, i] = 0;
        m_voxelizer.Voxels[(size / 2) - 1, (size / 2)     , i] = 0;
        m_voxelizer.Voxels[(size / 2) - 1, (size / 2) - 1, i] = 0;
}
```

This image on the above was generated by running the shown simple 6 lines of code. This code would be even shorter if the center of the sphere wasn't a 2 x 2 block.

Marching Cubes



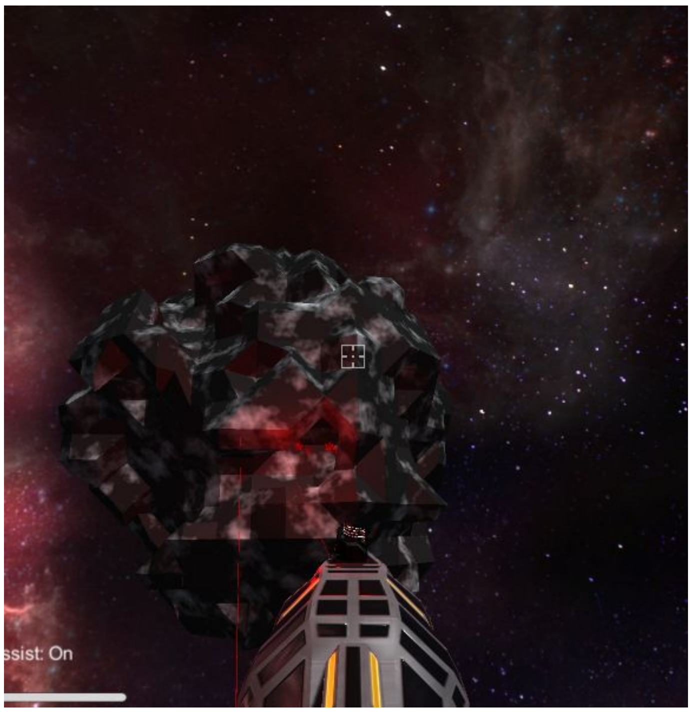


Marching cubes is an algorithm that takes voxels and turns them back into a triangle based mesh. These two images are the results of running Marching Cubes on both our voxelized sphere and our voxelized sphere with a hole bored through. While these may not be the detailed spheres we started with they do look a lot more like asteroids you might find in a game.

We did attempt to improve our asteroids further by implementing a Dual Contouring algorithm, which would have provided even better detail and surface smoothing, however we did not have time to get a solution working.

Perlin Noise Shader

We modified a Perlin Noise shader used for texturing the asteroids by merging it with a Unity lighting example shader. The noise shader itself was quite interesting, as it precomputed a 3D noise gradient texture by storing coordinates in the rgb channels and used uv mapping to retrieve and interpolate 3D noise at a given point. We combined this with a multi-light Unity shader which calculated ambient, diffuse, and specular light, and multiplied the resulting color output of the lighting calculations by the result of the noise in the fragment shader. This allowed our laser beams to cast light on the Perlin Noise textured asteroids, which results in an impressive effect. (shown to the right)



Customizable Asteroid Field

The asteroid field is generated in two phases. The first uses NBodySim code which simulates physics on bodies in a compute shader. We give the player control over placing a bounding sphere within the simulation space. This sphere's center origin point will specify the 0,0,0 location in the game scene. Every asteroid is distance checked from the bounding center, and any out of range are culled. Those remaining have their world positions converted to be relative to the new origin. This data is persisted, the new scene is loaded, and an AsteroidGenerator script retrieves the positions (now relative to the game scene origin), scales them by a constant factor (or else the distance between them would be much too small), and merges nearby points (by a specified threshold). Each asteroid is generated using the above specified algorithm by a thread pool (which has a configurable size in the settings). There is also a bounded maximum number of spawnable objects (for performance reasons, this is also modifiable in the settings), although we never hit that point in testing. After each asteroid is generated, it is spawned at it's scaled scene position, with a random rotation, random rotational force, and slightly randomized scale. The scale is also affected by the number of bodies merged in the previous steps (more merged bodies scales the resulting asteroid linearly). One thing to note is that the asteroid field is derived directly from the player selection in the simulation part of the game. Therefore, it is possible that asteroids will spawn over the player, resulting in a so called "bad start". The easiest way to resolve this is to try generating a new field. One final note is that, for convenience of demoing, we always spawn one asteroid right in front of the player at game start. This is in case all subsequent asteroids are spawned extremely far from the player (a sparse simulation selection), and in general for testing our implemented features.

The Ship

The ship was constructed out of a combination of free assets and script examples. The most notable parts are the ship controller, and the three subsystems, which will be described in more detail in the next three parts. The controller itself was left mostly unchanged, although the key bindings were changed and features were added to the original source.

Ship Laser

The laser is rendered using Volumetric Lines from the Unity Asset Store, and has a modulated pulse over time which is also lerped by size and distance on start-up and cool-down. It performs raycasting along it's forward vector when on and, if it hits an asteroid, will pass the hit information along so the asteroid can update it's voxel data and subsequently it's mesh.

Ship Radar/Scanner

The ship has an onboard radar which uses a really cool image effect shader. It uses the depth buffer and interpolated rays to the far clip plane to reconstruct world space positions. The author has a great video on the topic here: https://www.youtube.com/watch?v=OKoNp2RqE9A

Ship Turrets

We implemented two constrained-axis turrets from examples in the Unity Asset Store. These turrets rotate on two axes to aim at a world-space coordinate. They fire on fixed intervals at the nearest asteroid, producing small laser beams rendered using Volumetric Lines from the Asset Store. Each will fire in a straight path at the nearest point on the bounding box of the mesh collider attached to the targeted asteroid. Unity does not directly support finding points on a mesh collider, so the bounding box is an effective approximation. On collision, they will deform the mesh equivalently to the mining laser.

Parallelism and Multithreading

Our project featured numerous uses of parallelism. We utilized a multithreading library for Unity Coroutines called ThreadNinja. This allowed us to parallelize the spawning of asteroids via use of a thread pool. Each thread is designated a set of asteroids, and sequentially spawns one at a time, each taking several seconds to generate. The other main parallelism lay in the NBodySim code we included for world generation. It was implemented using a compute shader which ran on the GPU to simulate thousands of independent bodies interacting realistically. We were able to execute this code and then copy out positional information from the ComputeBuffer to generate a localised point cloud around a specified origin, as was discussed in the asteroid field generation section above.

References

- Marching Cubes: https://github.com/Scrawk/Marching-Cubes
- NBodySimulation: https://github.com/Scrawk/GPU-GEMS-NBody-Simulation
- Mesh Voxelizer: https://github.com/Scrawk/Mesh-Voxelization
- Radar Scan Easing:
https://github.com/lordofduct/spacepuppy-unity-framework/blob/master/SpacepuppyBase/Twee n/Easing.cs
- Ship Camera: https://github.com/densylkin/RTS_Camera
- Engine Sound: https://freesound.org/people/qubodup/sounds/146770/
- Wireframe Shaders: https://github.com/Chaser324/unity-wireframe
- Ship Model: https://www.assetstore.unity3d.com/en/#!/content/82240
- Skybox:
https://assetstore.unity.com/packages/2d/textures-materials/sky/spaceskies-free-80503
- Music: https://assetstore.unity.com/packages/audio/music/zero-gravity-7398
- General Sound Effects:
https://assetstore.unity.com/packages/audio/sound-fx/epic-arsenal-essential-elements-demo-pa cks-38428
- Laser Sound Effects:
https://assetstore.unity.com/packages/audio/sound-fx/weapons/laser-construction-kit-15966
- Laser Beams:
https://assetstore.unity.com/packages/tools/particles-effects/volumetric-lines-29160
- Turret Control 1: https://assetstore.unity.com/packages/tools/ai/simple-turret-system-92160
- Turret Control 2: https://assetstore.unity.com/packages/tools/free-turret-script-56646
- Ship Main Gun:
https://assetstore.unity.com/packages/3d/characters/robots/enemy-turrets-27858
- Ship Controller: https://github.com/M4deM4n/ShipController
- Scanner Shader: https://github.com/Broxxar/NoMansScanner/
- Multithreaded Coroutines: https://www.assetstore.unity3d.com/en/#!/content/15717
- Free Camera: https://github.com/ttammear/unitymcubes
- Perlin Noise Shader: https://github.com/Scrawk/GPU-GEMS-Improved-Perlin-Noise
- Multiple Light Shader Example:
https://en.wikibooks.org/wiki/Cg_Programming/Unity/Multiple_Lights
- Procedural Perlin Noise:
https://www.digital-dust.com/single-post/2017/03/14/Procedural-noise-in-Unity
- Unity Standard Assets: https://www.assetstore.unity3d.com/en/#!/content/32351