Christina Obereigner

Visual Computing Assignment

1 MARCHING CUBES

Used technology: Unity 2021.3.1f1

The following github repository was used for the marching cubes assignment:
https://github.com/Scrawk/Marching-Cubes-On-The-GPU

The project was extended by adding camera controls and a wireframe mode that can be enabled and disabled on the camera move component. All other implementations were used from the github repository and are NOT self-implemented. I spent most of this assignment trying to understand how the implementation works.

The main entry point of the program is the MarchingCubesGPU.cs file. At the beginning the three buffers that are needed for the marching cubes are created. Those are the noise buffer, used for the generated voxels of the perlin script, the normals buffer, which holds the normals of the voxels and the mesh buffer, which holds the vertexes that are generated by the marching cubes compute shader. In the first step the perlin noise is generated via the perlin noise .cs spript. Afterwards the tables from the perlin and some additional settings are sent to the perlin noise shader. The shader then creates the voxels from the input data. Afterwards the voxels are sent to the normals shader, which calculates the normals for the voxels. In the last step the mesh vertices are generated in the marching cubes shader. The result can then either be rendered in OnRenderObject() via a material with a draw buffer shader and Graphics.DrawProceduralNow() or used to create Mesh that is than converted to game objects.

1.1 SHADER

1.1.1 MarchingCubes

The MarchingCubes shader is responsible for creating the actual mesh. In the beginning a normal cube is created. Note that the border vertexes are ignored as they don't have any neighbours. For each vertex of the cube it is checked if it lies inside or outside of the surface. Once this is know the information can be used to find the edges that intersect with the surface inside the edge table. Afterwards the exact point of intersection is calculated for each edge. In the last step the triangles that are needed for the surface are saved. A triangle connection table is used for this step to determine if the triangles are valid.

1.1.2 PerlinNoise

The Perlin Noise shader is used to generate voxels from the perlin noise that was previously created in the GPUPerlinNoise.cs. Perlin is a type of gradient function that is generally used to create procedural textures and in this case generates the surface that should be displayed.

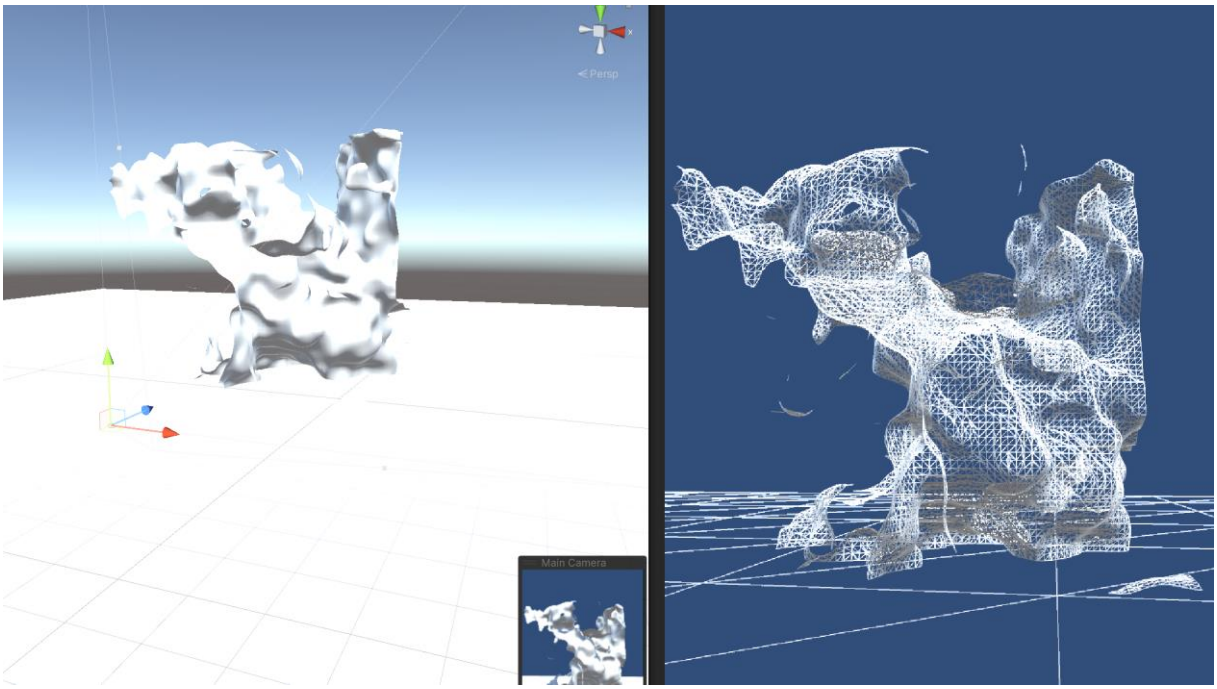
1.1.3 Normals

The normal shader is a compute shader that calculates the normals from the noise buffer.

1.1.4 DrawStructuredBuffer

This shader is used for the material to render the marching cubes mesh in OnRenderObject() method. The shader itself is only setting the vertex position and color because the actual drawing of

the points is handled by Unity itself via `Graphics.DrawProceduralNow(`MeshTopology.Triangles, SIZE).



2 REAL-TIME RAY TRACING

Used technology: DirectX 12, Visual Studio

At the beginning of the program the pipeline and the needed assets are loaded. For displaying a cube instead of a triangle the vertexBuffer inside LoadAssets() was given additional vertices to create a cube.

In the second step the acceleration structures are created. At first the bottom level structures for the cube and the plane are created. Out of those the two instances (cube and plane) are set and the top level AS are created for each instance.

The raytracing pipeline is created as follows: At first all needed shaders (RayGen, Miss, Hit,…) are compiled and added to the pipeline. Afterwards a Signature for each shader is created. The signature defines which parameters and buffers will be accessed, for example the Hit and Miss shaders don't need any additional resources while the RayGen shader needs the raytracing output and the top-level acceleration structure. The hit groups and root signatures are defined. At the end the max payload and max attribute size as well as the max recursion depth is set and the pipeline is generated for execution on the GPU.

Because the scene contains multiple instances a per instant constant buffer is created. This can be used to set different colors for the different instances. After that also the raytracing output buffer and the camera buffers are created.

At the end the shader resource heap and the shader binding table are generated. The shader binding table is where all programs are bind together in order to know which program to execute. The hit shader is added for each instance in the scene.

2.1 SHADER

2.1.1 RayGen

The RayGen shader is the main shader of this assignment. In here the ray for the raytracing is created. The ray is started at the location of the camera and launched in the view direction. At the end TraceRay() is called which then actually shoots the ray through the scene.

2.1.2 Hit

The hit shader is used when the ray hits the closest object. The lighting and shadowing are calculated inside this shader. In this case the closestHit method is called once for the cube object and once for the ground plane.

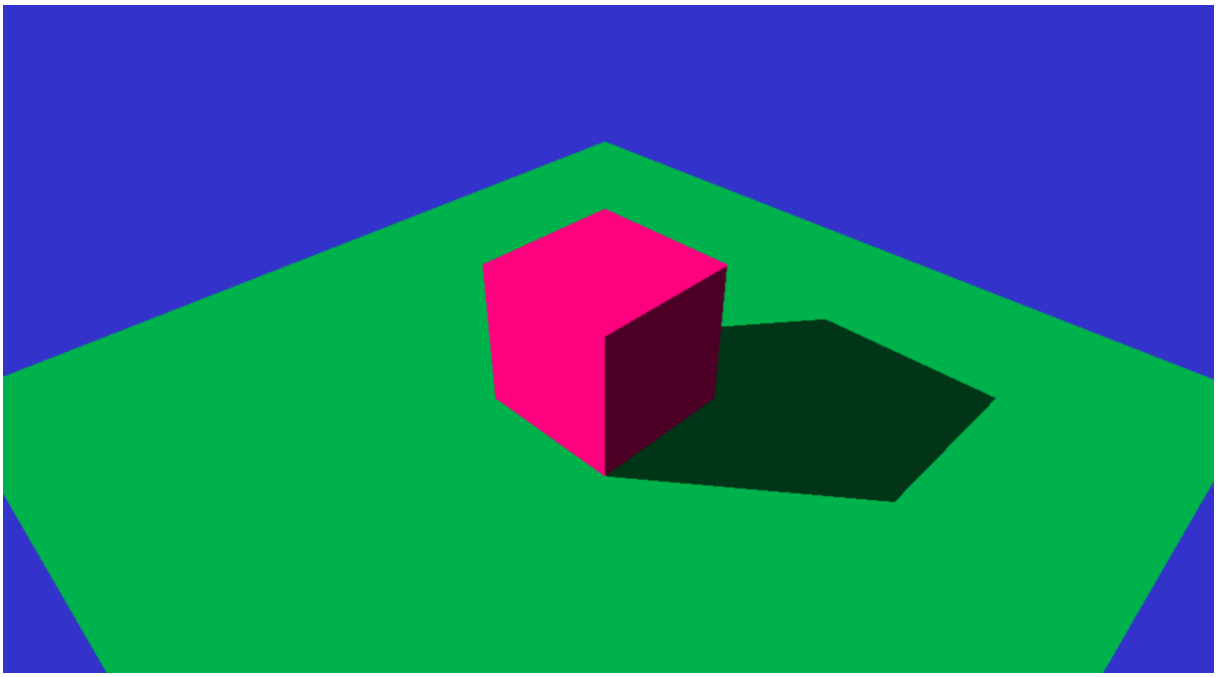
2.1.3 Miss

The miss shader is the simplest of the three used shaders. It is used when the Ray does not hit any objects and simply sets a default color in that case.

The following Tutorials were used for this assignment:

- https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-1
- https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-2
- https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial/extra/dxr_tutorial_extra_perspective

- https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial/extra/dxr_tutorial_extra_per_instance_data
- https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial/extra/dxr_tutorial_extra_another_ray_type



3 Point Cloud Rendering

Used technology: Unity 2021.3.1f1

I tried two different implementations in Unity for the point cloud rendering. The first is a (for Unity) typical shader implementation while the second one is a simpler way to display point clouds in Unity via VFX graphs.

In both implementations the data is parsed by simply reading the model.pts file via System.IO.File.ReadAllLines(). Each line is then split by the whitespaces. The position coordinates are saved in a Vector3 array and the Color values are saved in a Color array.

3.1 Shader Implementation

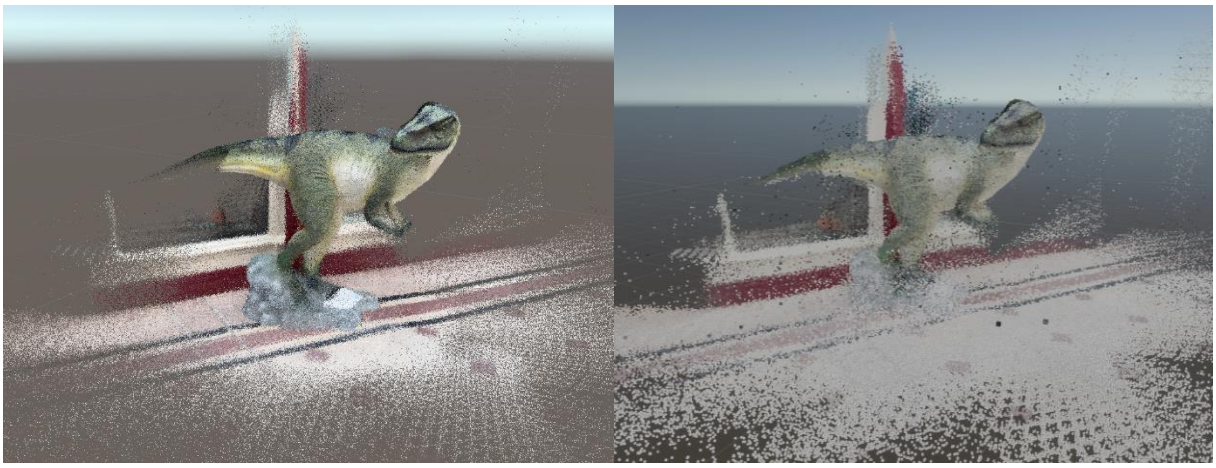
For the shader implementation two compute buffers (one for the positions and the other for the colors) are created. Both buffers are then sent to the material shader. The shader itself is only setting the vertex position and color because the actual drawing of the points is handled by Unity itself via `Graphics.DrawProceduralNow(MeshTopology.Points, number, 1).`

The following Source was used for this implementation: https://forum.unity.com/threads/point-cloud-compute-shader.764363/

3.2 VFX Implementation

The second implementation is not so much a visual computing implementation but rather an easier way to display point clouds in unity. For this a Visual Effect Graph was created that spawns a stationary cube particles at a given position with a given color. The size of the points can be altered in this implementation.

The following Tutorial was used for this implementation:
https://www.youtube.com/watch?v=P5BgrdXis68



Left: shader implementation, right: VFX implementation

4 NEURAL RENDERING

Used technology: Instant NGP (https://github.com/NVlabs/instant-ngp), COLMAP (https://github.com/colmap/colmap)

For this assignment I followed the following video tutorial to setup instant ngp: https://www.youtube.com/watch?v=kq9xlvz73Rg

For testing purposes I took 127 images of a plush llama. COLMAP was then used to do the pose estimation/reconstruction and generate the transform.json for the image set. When the generation of the transform.json is done the learning and rendering can be started by calling “instant-ngp\build\testbed.exe --scene data/llama”. One issue I noticed was that the render of the 3D llama was rotated by 90 degrees. I assume this is because I took the images with my smartphone in portrait mode. I tried to fix this by rotating the images to the side but this didn’t fix the issue.