3D Object Rasterization and Rendering with CUDA - Final Report

Minyu Cai (minyuc), Jinyan Zhu (jinyanz)

webpage: `https://jerrycmy2001.github.io/3d-object-rendering/`
Code: `https://github.com/jerrycmy2001/15618-team-project`

1 Summary

We implemented 3D object rasterization to render a 360-degree camera view of the object. We demonstrated how we can achieve great speedup by parallelizing the task among multiple GPUs with CUDA and OpenMP.

2 Background

2.1 3D Object Projection

In order to display the 3D objects from the perspective of the camera, the 3D objects need to be first projected to a plane. The input of this algorithm is the x, y, z coordinates of the triangle vertices in world space, and the output would be the coordinate on the plane, as well as the z value (for z-buffering to determine which object is at the top). This algorithm is essentially a matrix multiplication, where for each vertex (x, y, z), we calculate the vector multiplied by the camera matrix and projection matrix.

This workload can be parallelized because each vertex can be calculated independently. Meanwhile, the camera matrix and projection matrix is shared data among all of the computations.

In terms of data structures, we let the vertices to be a float array with length $3 \times 3 \times N$, where N is number of triangles. Therefore, the x, y, z coordinates of the three vertices of an triangles is laid out sequentially in memory.

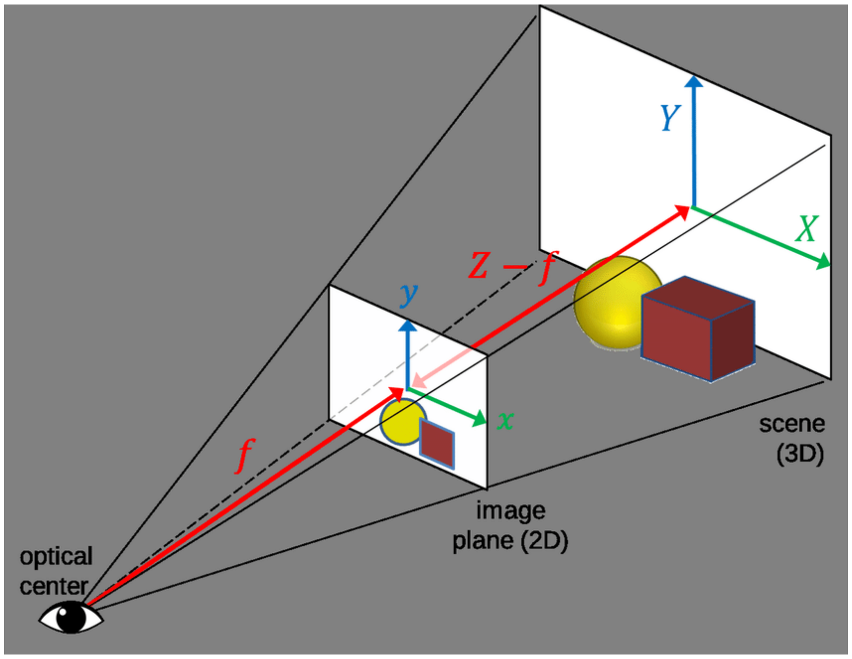

Figure 1: 3D Object Projection

2.2 Rasterization

We run the rasterization algorithm to determine whether a pixel on the image lies within the boundary of a triangle. Given the projected coordinates of the triangle vertices, we use the Barycentric Coordinate Method to check if the pixel is inside, and if yes, the weighted z value of this point. After iterating through all triangles, the displayed color of this point is the triangle with the least z value, if any.

This algorithm can be parallelized among all pixels, as the calculation of multiple points on the plane is data independent. The coordinates of triangles are shared data among all computations in this scenario.

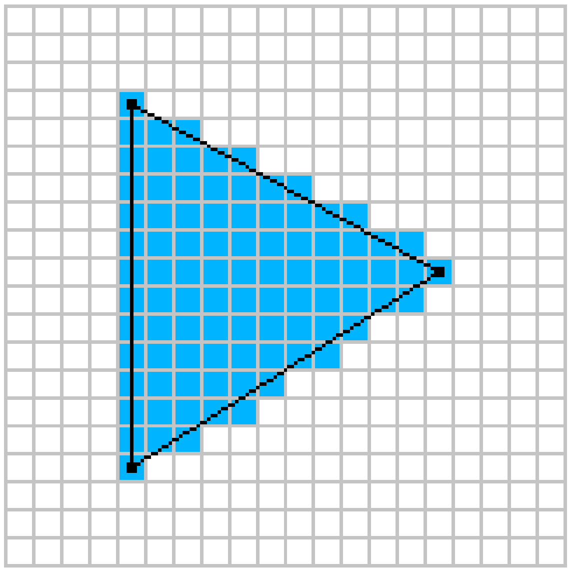

Figure 2: Rasterization Task

3 Approach

3.1 System Overview

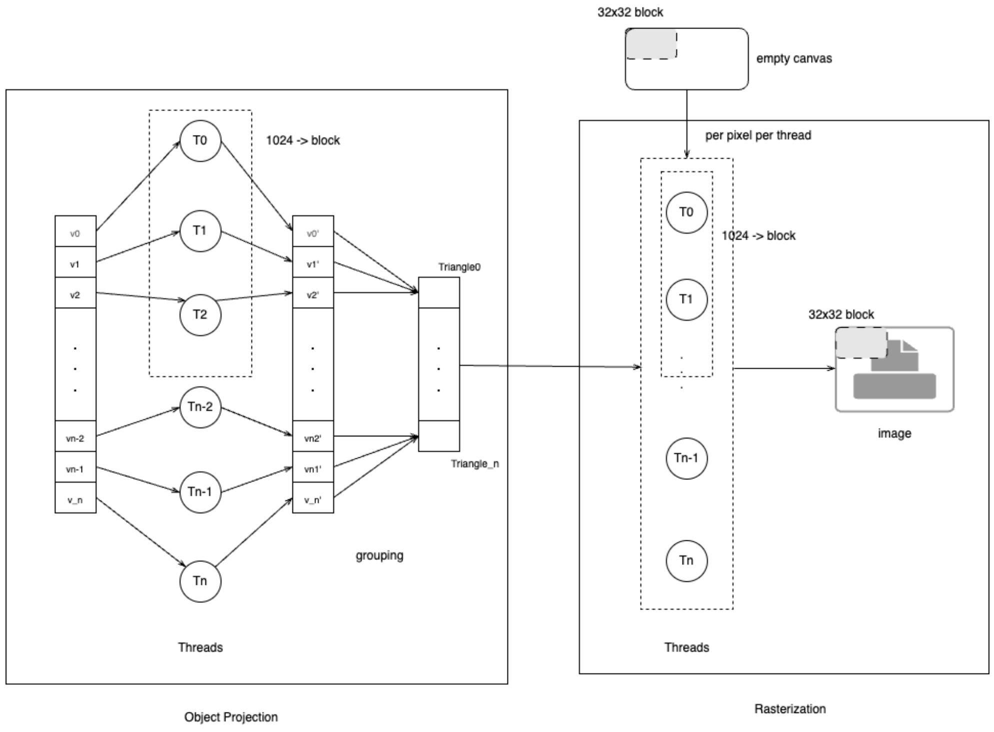

Figure 3: System Framework

3.2 Tools and Frameworks

We used C++ for the serialized implementation on CPU, and used CUDA to parallelize the task on GPU. Furthermore, we used OpenMP to distribute the task among multiple GPUs.

Meanwhile, we built our code upon the project 2 handout so that we don't need to worry much about image display.

3.3 3D Object Setup

We model 3D objects as an abstract class with an abstract method that returns the list of triangles that construct the object. Built upon this, we implemented simple 3D objects like cubes and tetrahedrons, which can be laid out to create larger and more complex 3D objects.

3.4 Parallized Algorithm Implementation

3.4.1 3D Object Projection

In our CUDA implementation, we allow each CUDA thread to handle the matrix multiplication of a single vertex, and launched multiple blocks to do the calculations of all the triangles. This approach also takes advantage of the locality of our data structure where vertices are coalesced in the array.

3.4.2 Rasterization

Since rasterization is data independent on the pixels, we naturally assign each CUDA thread to a pixel, and launched multiple 2-dimensional blocks to cover all pixels on the image. We also loaded projected vertices into shared memory to further optimize the computation.

3.4.3 Task Distribution among Multiple GPU

Eventually, we used OpenMP so that we can take advantage of multiple GPUs when doing the computation. We segmented the image in height according to the number of GPUs we have, and assign each GPU to a part of the image. Note that we'll need to allocate the arrays, as well as move data structures to and from multiple GPUs.

4 Results

4.1 Scene Demo

We first build a few basic 3d objects by serializing them to triangles for rendering:

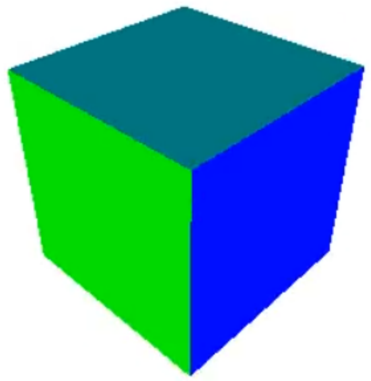

Figure 4: 3D Cube

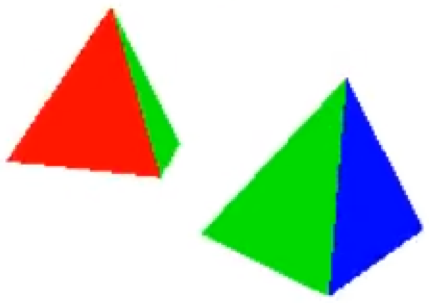

Figure 5: 3D Tetrahedron

Then based on those simple 3D objects, we're able to build more complex 3D scenes:

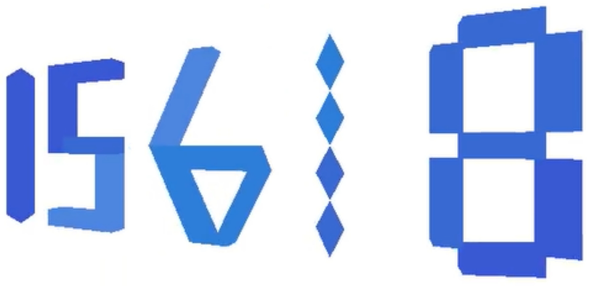

Figure 6: Course Title Modeled in 3D Objects

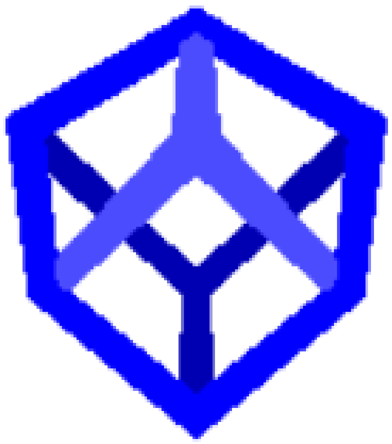

Figure 7: Cube Visual Illusion Modeled in 3D Objects

4.2 Speedup Analysis

We measure the overall time it takes for the algorithm to run on different platforms, namely CPU, single GPU, 2 GPUs, and 4 GPUs. Then we can calculate the speedup of different parallel methods. Note that we use Tesla V100-SXM3-32GB for performance tests.

Meanwhile, we set up the scenes so that it fully tests the rendering functionality, and the complexity increases incrementally. Single, double, non-orthogonal, and square are just 1 or 2 triangles in scene. Cube and tetrahedron scenes are made up of around 10 triangles. Rand$N$ means N randomly generated cubes or tetrahedrons scattered in the scene.

Scene	Core Type	Core Num	Total operating time(ms)	Speedup
Single	CPU	1	10.5157	1x
Single	GPU	1	10.0288	1.050x
Single	GPU	2	7.6928	1.367x
Single	GPU	4	7.4083	1.419x
Double	CPU	1	14.8172	1x
Double	GPU	1	10.3929	1.426x
Double	GPU	2	7.7442	1.913x
Double	GPU	4	7.4937	1.977x
Non-orthogonal	CPU	1	18.6029	1x
Non-orthogonal	GPU	1	10.1933	1.825x
Non-orthogonal	GPU	2	7.4329	2.503x
Non-orthogonal	GPU	4	7.8168	2.380x
Square	CPU	1	11.8272	1x
Square	GPU	1	11.1059	1.065x
Square	GPU	2	7.4781	1.582x
Square	GPU	4	7.7955	1.517x
Cube	CPU	1	53.5763	1x
Cube	GPU	1	10.5218	5.092x
Cube	GPU	2	7.3611	7.278x
Cube	GPU	4	7.5808	7.067x
Tetrahedron	CPU	1	38.4358	1x
Tetrahedron	GPU	1	10.2509	3.750x
Tetrahedron	GPU	2	9.3620	4.106x
Tetrahedron	GPU	4	7.3248	5.247x
Rand8	CPU	1	379.4424	1x
Rand8	GPU	1	12.5828	30.162x
Rand8	GPU	2	12.1200	31.271x
Rand8	GPU	4	13.7175	27.644x
Rand27	CPU	1	1062.7054	1x
Rand27	GPU	1	11.6280	91.394x
Rand27	GPU	2	14.8206	71.707x
Rand27	GPU	4	16.1574	64.758x
Rand64	CPU	1	2775.2317	1x
Rand64	GPU	1	14.1171	196.669x
Rand64	GPU	2	16.2199	171.085x
Rand64	GPU	4	13.6871	202.851x
Rand125	CPU	1	7054.1004	1x
Rand125	GPU	1	17.1804	410.594x
Rand125	GPU	2	15.0593	468.424x
Rand125	GPU	4	10.2023	691.689x
Rand343	CPU	1	27253.5530	1x
Rand343	GPU	1	30.8631	884.838x
Rand343	GPU	2	18.2113	1496.488x
Rand343	GPU	4	11.3318	2405.002x
Rand1000	CPU	1	90018.5799	1x
Rand1000	GPU	1	75.8732	1186.477x
Rand1000	GPU	2	41.4353	2172.774x
Rand1000	GPU	4	26.9137	3344.691x

Table 1: Speedup of different scenes on PSC machines

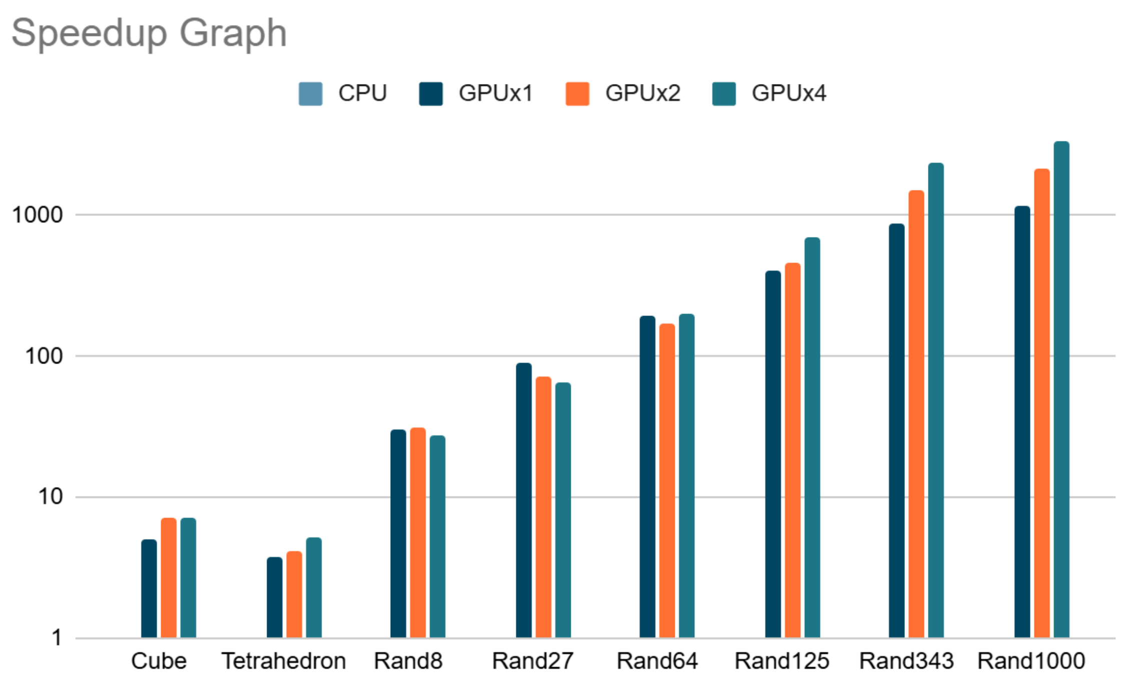

Figure 8: Speedup Graph

Above are the experiment results with different scenes. We mainly focus on the speedups with different core types. We can notice that

- Experiments on GPUs have less operating time than those on CPUs.
- More GPUs lead to less operating time.
- With larger scene setting, the speedup is greater and more obvious when using GPU.

The above observation makes sense as GPUs have more computing power and more GPUs can distribute the workload and reduce the operating time.

However, we also find the bottleneck of the system. With simple scenes like single triangle, square, and etc., the operating time with 4 GPU cores gets stuck around 7.5 ms. This is because when the scenes are simple, the computation time is very limited and the main bottleneck of the workload is the memory transfer of vertices, colors, and the image. Note that for each render operation, we'll need to transfer the vertex data from CPU to GPU, allocate the image on GPU, and transfer back the image from GPU to CPU when the computation is done. This is where we believe takes most of the time in small scenes.

When the scene grows more complex, for example 1000 random 3d objects, the speedup can grow to more than 1000x, and we get close to perfect speedup for multi-GPU parallelization. This is because we're fully parallelizing the projection and rasterization workload. Since there're so many triangles, doing the projection for each vertex, traversing the triangles and calculating the color of each pixel becomes very computationally intensive. With GPU and CUDA, we can fully parallelize the task among multiple CUDA threads. With the help of shared memory to store data structures that're accessed by all threads, the speedup becomes quite large.

Note that the largest scene is big enough so that it is almost unreasonable to do on CPU (several minutes to render a single frame).

5 References

1. `https://joshbeam.com/articles/triangle_rasterization/`
2. `https://www.researchgate.net/figure/Diagram-illustrating-how-3D-object-in-the-environment-can-be-projec`
   `fig5_311969455`

6 List of Work and Distribution

6.1 List of Work

Minyu Cai: 3D object modeling, CUDA implementation and optimization, task distribution among multiple GPUs
Jinyan Zhu: serial implementation involving camera and 3D objects, camera rotation, scenes setup

6.2 Distribution

We contributed to this project fairly equally, so 50%-50%.