Final Project Proposal

Home

Title

Parallel Triangle Rasterizer (Wei-Ting Tang, Yiyun Wang)

URL

https://white123.github.io

Summary

We are going to create an optimized parallel implementation of a triangle rasterizer on the GPU platform, and perform a deep analysis of the performance characteristics.

Background

The triangle rasterization process is a common way of shading computer graphics. It is computationally intense because a high-quality computer image usually contains thousands and even millions of triangles, so computation efficiency becomes a matter. A rasterization pipeline includes the following steps:

1. Convert the vertices of the triangle to raster space
2. Iterate through all of the pixels and check if the pixel is covered by the triangle
3. Conduct the depth-buffer test and update the pixel attribute accordingly

We can conduct parallelism either across pixels or across triangles. Both of these strategies would benefit from GPU parallel computing, but it also creates extra costs of synchronizing the states and maintaining the dependency. We will introduce more details of the challenge in the below section.

Challenge

Unlike assignment 2, triangle rasterization could lead to a more unbalanced workflow. For example, to rasterize an extremely “skewed” triangle that locates from the bottom left to the top right, the rasterizer will iterate through almost all pixels, but the triangle only covers small proportion of them (i.e., the diagonal of the screen). That gives us more potential improvements if we can come up with a more sophisticated algorithm to determine the triangle coverage.

Moreover, triangles are not entirely independent of each other. A triangle can be generated from a bigger polygon, which means some of the triangles shares the same edge or boundary. It can cause duplicate emissions at the pixels located on the boundary. A potential solution is to use the top-left rule introduced by OpenGL to deal with this problem. However, it needs to split the triangle into upper and lower parts, and it increases the difficulty of parallelism in the perspective of synchronization bound and memory accesses.

Although the problem itself seems not too hard to parallelize, it is not that obvious to balance the work flow since it requires high memory accesses and synchronization.

Resources

We are going to develop a sequential version of the triangle rasterizer using the starter code and tutorial outlined here. We are also going to refer to course materials of 15462 Computer Graphics about implementation of rasterization pipeline and its optimization techniques.

When evaluating the performance of our implementation, we are going to use the GHC machine cluster.

Deliverables

Plan to achieve

- Implement a fully functional parallel triangle rasterizer
- Achieve 30 FPS rendering with 3840 x 2160 resolution and 10k triangles
- Speedup performance analysis and graphs of different number of triangles and resolutions

Hope to achieve

- Achieve 60 FPS rendering with 3840 x 2160 resolution and 10k triangles
- Scale well with higher resolutions and with millions of triangles (i.e., no significant performance drop-off)
- Real-time rendering animation for demo

Platform

C++ and Cuda are suitable for our implementation because we can easily use Cuda to conduct parallel computing on Nvidia GPU on GHC machines.

Schedule

Week	**Plan**
Week 1 (11/6 - 11/12)	Submit proposal and build up our benchmark sequence implementation
Week 2 (11/13 - 11/19)	Build a naive parallel implementation and analyze the bottleneck
Week 3 (11/20 - 11/26)	Build a more complicated version and finish milestone report
Week 4 (11/27 - 12/3)	Optimize our implementation to make sure it performs well on normally scaled inputs
Week 5 (12/4 - 12/10)	Handle the huge input data and deal with potential memory bound
Week 6 (12/11 - 12/17)	Analyze the implementation with different inputs and finish final report