Abstract

Render a 3D scene represented by a set of polygons and associated textures and material information and save result to an image file.

Introduction

A 3D scene can be represented, via a set of triangles which approximates the desired geometry. Material information, such as textures, lighting information, and more can further be associated with a particular triangle. The goal of the renderer is to create a single image which represents what a camera setup would capture of the given scene.

To accomplish this, a matrix can be created which represents the camera information (both position/rotation information, and a projection transformation from 3D to 2D). This matrix can be used to transform each triangle to 2D space. From this 2D triangle, we can obtain the set of relevant pixels of our output texture. Depth information is obtained from triangle vertices and linearly interpolated for each pixel in the output image. A "shader" function can take relevant triangle information to determine appropriate colors for each of these pixels, depth information can be stored in an additional buffer which is used for conditional writing to the output buffer, such that the order in which triangles are drawn is not important.

Once all triangles in the scene are rendered, post processing may be performed on the output image to add additional screen space effects. These operations are performed per pixels. Further effects can also be applied via including additional data buffers and information attached to each vertex of a triangle.

Once all operations are complete, the image buffer is saved to a file on disk as output.

Parallelization

The real world application of this technique is for real time 3D rendering, most notably in the context of video games, and other applications which require real time interaction. Real time interaction requires a total render time of about 0.0333 seconds per image minimum. This is accomplished via massive hardware parallelization on the GPU. Each vertex is independently handled by one of thousands of cores on the GPU, further cores have dedicated hardware for performing fast matrix and vector operations. When writing a software renderer I will have access to only comparatively fewer cores and will not have dedicated linear algebra hardware. This provides many options for a parallelization strategy.

The first and most simple would be to render each triangle on its own thread. Synchronization would be necessary when writing to the output image as multiple triangles can cover the same point. This should provide a speedup compared to running in serial however I do not thing this will produce the best results. The work can be divided into sections which are performed by threads (for example, vertex transformations are all performed together in one step), once a thread completes a transformation for a vertex, rather than moving on to rendering the corresponding triangle it will instead transform another open vertex. Once all vertices have been transformed, the renderer will move into a new stage and perform those operations in parallel. Required vector and matrix operations can also be performed in parallel and I can experiment with performing these in serial vs parallel in various scenarios. For example, another possible implementation might be a completely serial approach, but with all matrix/vector operations parallelized.

References

Mathematics for 3D Game Programming and Computer Graphics
Eric Lengyel, 2002

Tiny Renderer
Dmitry V. Sokolov, Université de Lorraine
A series of articles, with corresponding C++ code, implementing a serial 3D renderer in software.

Common Mistakes in OpenMP and How To Avoid Them - A Collection of Best Practices
Michael SüB and Claudia Leopold