Real-Time Rendering and 3D Games Programming: Custom Rendering Engine

Søren V. Poulsen*
M.Sc. Stud.
DTU Compute, Lyngby

Geoff Leach†
Lecturer and Coordinator
RMIT University

Tim Mutton‡
Head Tutor
RMIT University

Figure 1: *Crytek Sponza scene*

Abstract

This paper describes the design and implementation of a custom engine for rendering various 3D scenes using OpenGL. Using modern rendering techniques, features such as dynamic point lights, screen-space ambient occlusion and deferred rendering has been implemented in a GNU/Linux C++11 application with minimal use of external libraries. The project has been completed as an assignment in the course *COSC1224/1226 Real-Time Rendering and 3D Games Programming*[1] coordinated by professor Geoff Leach of RMIT University.

The project has been completed in 12 weeks during semester 2 of 2013 by M.Sc. Stud. Søren V. Poulsen of DTU Compute in Lyngby, Denmark.

October 2013

1 Introduction

The purpose of this project assignment has been to demonstrate skills and knowledge of various OpenGL features. Instead of aiming for visual fidelity or high frame rate efficiency

2 Features

Below is a list of the features that are currently part of the rendering engine in a working state:

- Fully deferred rendering
- Color(diffuse) texturing and bump-mapping
- Dynamic and (theoretically) unlimited point light sources
- Screen-space ambient occlusion
- Efficient CPU-based ray/scene intersection test using bounding volume hierarchies

3 Implementation

The application has been implemented without use of a preexisting framework or rendering library.

*e-mail: svipo@dtu.dk
†e-mail:gl@rmit.edu.au
‡e-mail:tim.mutton@rmit.edu.au

[1] `http://goanna.cs.rmit.edu.au/~gl/teaching/rtr&3dgp/`

3.1 Deferred Passes

Deferred rendering is a rendering technique that separates geometry and lighting operations into multiple passes. Using this technique has the great benefit of reducing the performance hit of having many lights in a scene regardless of the complexity of the scene geometry.

In the first pass (the geometry or G-pass), information about the scene geometry is gathered into multiple buffers (which in turn are simple textures). In this application the texture color (albedo) is stored in a 4×8 bit texture buffer, the surface normal in a packed 2×16 bit floating point (FP) texture buffer, and finally the depth and stencil in a $24 + 8$ bit FP/integer texture buffer.

Generating images in this application happens through multiple rendering passes. The first pass

3.2 Lighting

3.3 Ray-tracing

In an effort to support basic game-like features such as shooting weapons, the application is capable of doing ray-scene intersection. That is, given a starting point and direction, determine the closest point (if any) of intersection with the triangle meshes that makes up the scene. A typical scene has many triangles, so a naive $O(n)$ search is deemed to be too inefficient when scenes become non-trivial.

Instead a bounding volume hierarchy (BVH) of axis-aligned bounding boxes (AABBs) containing triangles is built in the initialization stage of the engine. The octree data structure has been selected as the BVH structure. When querying for ray-triangle intersections, the top level (root) AABB of the BVH tree is tested for intersection with the given ray. If there is such an intersection, the 8 children AABBs beneath the root is tested and so forth until only leaf nodes are left. The leaf nodes contains simple triangle lists, which are then tested against the given ray.

In average- and best-case scenarios this method significantly reduces ray-triangle intersection tests and makes even large amounts of ray-scene queries for each frame feasible.

3.4 Asset Handling

Any rendering or game engine requires handling of assets such as shaders, textures, models, scripts, etc. The efficiency of the asset handling mechanics is a determining factor for loading times and/or memory usage, which makes it an important and essential part of any rendering engine.

This application uses a simple string-based caching mechanic. Each asset is given a name, which is stored as a simple textual string, that in most cases is directly related to the file path of the asset. When an asset is requested for usage (e.g. a model or texture file) the name of the requested asset is looked up in the asset cache. If found, a reference to the already loaded asset is returned, leaving the file system untouched. If the name is not found, then the asset is loaded from the file system and entered into the cache system.

Having all assets handled by the same asset management system has other benefits besides reduced file system utilization. For example, this application features hot-loadable assets meaning that assets that are altered by an external application are almost instantaneously reloaded in the engine. This has been accomplished using the Linux kernel subsystem *inotify* and has proven to be very beneficial while debugging the application.

4 Results

Bla.

5 Conclusion

Bla.