Advanced Game Engine

Ian Lilley and Sean Lilley

Advisor: Norman I. Badler

University of Pennsylvania

ABSTRACT

Game engines are complicated. They require up-to-date knowledge of many areas of real-time rendering as well as physics, sound, and animation. Although it is usually straightforward to implement individual components of an engine, putting everything together elegantly can be a challenge. Our project will focus primarily on implementing the newest techniques in several areas of real-time rendering, including cascaded shadow maps, advanced frustum and occlusion culling, forward plus rendering, and more. Finally, we hope to engage with the graphics community by making this project open source.

Project Blog: http://gameengineers.blogspot.com/

1. INTRODUCTION

Even the simplest of games needs a functional game engine. There are three components to a good game engine: speed, adaptability, and simplicity. A game engine must be fast. It must be optimized at all levels, especially for graphics and scene management. If a game has too much lag, users will stop playing it. Next, a game engine must be adaptable. It should be able to gracefully handle many different objects, scenes, and inputs. Finally, it should be simple. And artist or designer should be able to add things to the game without worrying about the low level details. Our goal is to make a game engine that incorporates these three qualities.

Making a game engine is an interesting challenge because it forces us to combine the best techniques from many areas of real-time rendering into one package. Although individual components of an engine make for good tech demos, there is value in putting them all together while maintaining high quality and performance. Finally, the source code for most popular game engines is not publicly available, so it would be good to expose these techniques to the graphics community in our own public repository.

The first step towards accomplishing these goals is to research the newest advancements in real-time rendering, specifically forward plus rendering (discussed later). We must design a system that makes it easy to add these new features to ones we have implemented in previous projects. By the end we hope to have a robust game engine that performs well and incorporates the best aspects of game rendering.

1.1 Design Goals

This project is geared towards the game development community. Game programmers benefit because they will be able to read our code and see how we organize the different structures as well as get a glimpse of some interesting graphics techniques. Game artists and designers will benefit because we will develop tools for fast content creation.

1.2 Projects Proposed Features and Functionality

A game engine is composed of many parts. Some of the features we would like to target (in no particular order): character animation, transparency, cascaded shadow maps [DIM07][EIS11], rigid body physics, asset loading pipeline, materials and BRDFs, object instancing, frustum and occlusion culling [RAK11], level of detail, reflections and refractions, bump mapping [LEN11], post processing effects (like motion blur, depth of field, and ambient occlusion) [MOL08], game event system, level editor, user interfaces, audio, multiplayer/networking, and forward plus rendering [HIR12]. In general, these features do not depend on each other so we have the freedom to decide the order in which we implement them. A parallel goal of this project is to always render above 60 fps for complex scenes on our target hardware.

2. RELATED WORK

We are largely inspired by the amazing results of modern game engines like Unreal Engine 4 and CryEngine 3. Although we do not have the manpower to match the output of those companies, we have access to a lot of their techniques through books and papers. We will be consulting books like *Real Time Rendering* [MOL08] and *Mathematics for 3D Game Programming and Computer Graphics* [LEN11] as well as SIGGRAPH demos and papers. Also, companies like AMD and NVIDIA are constantly creating demos for their new research. We plan take from all these different sources.

3. PROJECT PROPOSAL

We will be creating a game engine from scratch. In doing so we hope to accomplish two main goals: first, to explore the newest techniques that real-time rendering has to offer, and second, to make a game. This means not only writing a lot of code, but also thinking about game design and creating art assets.

3.1 Anticipated Approach

Here we discuss some of more interesting features of our game engine:

GPU accelerated object instancing, frustum and occlusion culling, and level of detail selection – Typical game environments are quite large and contain many instances of common objects like trees, rocks, enemies, etc. We propose a completely dynamic system that updates the number of instances to draw and their level of detail based on occlusion and frustum culling tests that are performed in a compute shader. Most engines perform these tests on a per-object basis on the CPU, with constant round trips to the GPU to get results from occlusion queries. Our GPU-only technique uses some of the newest features of OpenGL including compute shaders, atomic counters, and draw indirect buffers.

Forward plus rendering [HIR12] has the performance benefits of deferred shading without the high memory cost. In addition, forward plus rendering easily supports multi-sampling and transparency where deferred rendering does not. The steps for doing forward plus rendering are as follows: first do a Z-prepass of the scene. Next, use a compute shader to perform light culling and create light linked lists for each pixel or tile of the screen. Finally, render the scene again and compute the color of each pixel based on the lights in that tile and the material properties of the object.

3.2 Target Platforms

Languages: C++ and OpenGL 4.3

Operating Systems: Windows and Linux. No OSX because it is unlikely that they will have OpenGL 4.3 drivers by the time we do this project

Hardware: AMD and Nvidia OpenGL 4.3 compliant graphics cards

Code Libraries:

- GLFW/GLEW for handling OpenGL context and user input
- Bullet Physics for physics
- TinyXML for XML parsing
- GLM for linear algebra math library
- GLI for texture loading utilities
- OpenAL for audio
- Freetype for text

Software:

- Blender for 3D model creation, rigging, and animation
- Gimp for texture editing

3.3 Evaluation Criteria

We will compare the visual quality of our results against those of popular game engines. We will also be benchmarking our engine, timing different areas of code and fixing bottlenecks. Our goal is to stay above 60 fps for complex scenes.

4. RESEARCH TIMELINE

In order to complete most of the features in section 1.2, we will implement smaller features on a weekly basis and larger features on a monthly basis.

Project Milestone Report (Alpha Version)

- Complete all background reading
- Complete forward plus rendering and object instancing pipeline (essentially the core graphics engine)

Project Final Deliverables

- Implement as many features as possible from the list in section 1.2. In order to do this we will first complete the core rendering engine. With the remaining time we will implement some of the less critical features like UI, networking, and level editor.
- Tech demos showing off the various features separately and together.
- Open source repository.

Project Future Tasks

- Continue to add more features. A game engine is never complete because there is always new research on the horizon.

5. Method

The backbone of our rendering engine is a deferred pipeline. First the scene geometry is rendered. The fragment attributes (color, normal, specular, and depth) are outputted into the G-Buffer, which consists of color, normal, specular, and depth textures. Afterwards we perform a series of screen space effects including lighting, decals, reflections, and ambient occlusion.

Decals - projecting a 3d volume onto an area in the screen. This allows us to simulate bullet holes, footsteps, and other effects.

Reflection - for every reflective fragment, cast a 2D ray and traverse through the depth buffer to find an intersection point. Sample the color at this point to use as the reflective color.

Ambient Occlusion - For every pixel, send some feeler rays in screen space to determine the percentage of neighboring pixels that are intersected. A high percentage results in a darker appearance. Afterwards we blur the ambient occlusion texture to create a smoother look.

There are a variety of other rendering techniques that we implemented that are not directly involved with the deferred pipeline. This includes parallax mapping for simulating complex geometry on flat surfaces, specular mapping for added realism, skeletal animation through vertex-shader skinning, order independent transparency, cascaded shadow maps, and GPU instancing and culling.

Order independent transparency - Render the transparent geometry in a separate render pass and store transparent fragments in per-pixel linked lists. Afterwards, for each pixel sort and blend the fragments in the linked lists and alpha blend with the color texture in the GBuffer.

Cascaded shadow maps - in order to draw realistic shadows efficiently across large open areas. This technique involves constructing multiple shadow maps which expand across increasingly large areas in the world. In this way shadow quality adapts to the distance from the camera.

Rendering was the largest aspect of our game engine, but we explored other areas including physics, sound, user interface, and scene loading. To tie all our game subsystems together we use an event-driven component system similar to other game engines like Unity.

Supported physics features include rigid bodies, soft bodies, cloth, terrain, vehicles, and characters. One area where physics and rendering intersects is the creation of decals. When the user clicks the mouse, a physics ray is sent into the world and a physics object is retrieved. The decal cube is then added to the scene graph and transformed appropriately before being rendered by the decal manager.

For scene loading, we created a series of custom XML formats to suit our needs. This includes formats for meshes, materials, entities, and scenes. In addition we wrote some Python scripts for Blender to export content into our engine quickly. Our scene loading is sufficiently advanced enough that we no longer hard code any information in the engine itself, and even game behavior is stored in the XML files.

6. RESULTS

Although we did not perform very fine-grained benchmarking tests, we managed to stay above 60fps for all scenes with maximum visual quality. We noticed that the main bottleneck in the rendering pipeline is the screen-space passes, primarily reflections and ambient occlusion. Both of these effects handle about 30 operations per pixel which results in slower speeds the larger the window resolution. To counter these slowdowns, we give an option to lower the resolution of these effects to half or quarter the size, improving the frame rate significantly. On the other end of the spectrum, we did not observe any major performance hits from having many objects in the scene, unless there are hundreds of physics objects in a large pile (which is fairly uncommon in most games). This means we have a lot of freedom to make complex scene without worrying about bad performance. As far as we can tell, the GPU side of our application accounts for over 99% of the frame time.

Images from our application can be found inside the screenshots folder.

7. CONCLUSIONS and FUTURE WORK

Overall we are happy with the number of features we able to add over the course of the project. Although our focus was graphics techniques like deferred rendering, shadows, and animations, we managed to add a number of other features that make our program not just a rendering engine, but a full-featured game engine. Although each feature presented its own challenges, the greatest challenge of all was combining everything without compromising code quality and performance.

There is a lot more work to be done for this project. The main rendering feature we have not yet included is real-time global illumination. We wanted to hold off on this feature because there is a lot of new research in this area and it can be hard to gauge the performance-quality tradeoff just by reading papers. Next, we plan to add more post-processing effects like depth of field, motion blur, HDR, and tone-mapping for a more realistic look. On the content creation side, we plan on improving our level editor and object creation process with the ultimate goal of having a completely data-driven game, a necessity for artists and modders. One interesting approach might be to use a scripting language to accelerate in-game actions for characters and other game objects.

References

[HIR12] Takahiro Hirada, Jay McKee, Jason C. Yang: *Technology Behind AMD's Leo Demo* (GDC 2012).

[MOL08] Tomas Akenine-Moller, Eric Haines, Naty Hoffman: *Real-Time Rendering*. AK Peters, 2008.

[LEN11] Eric Lengyel: *Mathematics for 3D Game Programming and Computer Graphics*. Course Technology PTR, 2011.

[DIM07] Rouslan Dimitrov: *Cascaded Shadow Maps*. NVIDIA Corporation, 2007.

[EIS11] Elmar Eisemann, Michael Schwartz, Ulf Assarsson, Michael Wimmer: *Real-Time Shadows*. CRC Press, 2011.

[RAK11] Daniel Rakos: *Hierarchical-Z Map Based Occlusion Culling*, http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/