Deferred Rendering in a Multi-Threaded Component Based Engine

|  |  |  |
| --- | --- | --- |
| **Date:** |  | **February 18th, 2011** |
| **Authors:** |  | **Eric Moreau** |

# Abstract

In this paper, we will delve into the area of implementing multiple art styles and techniques in a deferred renderer in a component based game engine. We will also explore the benefits and detriments of using a deferred renderer and a component based engine together. In order to understand these two areas, we will briefly describe more standard structures and techniques to create a base of comparison. For deferred rendering, we will talk about the more standard forward rendering pipeline and talk about deferred rendering’s strengths and weaknesses in comparison to a forward renderer. For the component based engine model, we will compare against the hierarchal based engine model and highlight strengths and weaknesses for the component based engine. Along with this paper, we will illustrate his implementation attempt and analysis of his results.

# Introduction

In video game audiences today, players do not often consider what goes into creating the software that drives the games they play. However, one thing that is placed under a microscope is the quality of the graphics that are presented in these games. One of the pipelines that video game developers are starting to implement now is a deferred renderer. Deferred rendering is a style of pipeline that breaks down the rendering process into smaller parts and stored to be combined later. The one drawback to this type of system is that it limits the flexibility of the pipeline to one rendering style that must be applied to all graphical components. These components can include but are not limited to three dimensional meshes, particle systems, in-game heads up displays (HUD), and menu systems. The research and subsequent implementation attempt looks into this drawback and attempts to solve this drawback in hopes of adding the back some flexibility to the deferred rendering process.

# The Deferred Rendering Process

Deferred rendering is a shading technique in three dimensional computer graphics that makes use of modern hardware’s ability to render images to multiple targets and stored in a buffer to be combined later. [1] Common deferred renderers make use of a three pass process. The first pass takes a snapshot of what is to be rendered to the screen. This snapshot is called the G-Buffer and usually comprises of three maps; a diffuse map, a normal map, and a depth map. The second pass creates a light map using the normal and depth maps along with lighting information for that particular scene. The final pass combines the diffuse map created in the first pass and the light map from the second pass to create the final scene. The final scene is rendered to the screen and the process repeats itself.

The G-Buffer is created during the series of draw calls that the graphics device goes through on each loop. This is the only component of the deferred rendering process that is handled on a per draw basis. Both the light map and the final combine map passes are done right before the graphics device presents the final image to the screen. The light pass takes all the lights required for the scene and calculates how much a given pixel will be lit for the scene. The light map that is generated is a set of grey scale values that is added to the diffuse color later one. The final combine pass is the last step before the graphics device draws flushes the buffer to the screen to be drawn. Of the three passes, the final pass is the simplest since it is only combining two maps together.

## History

In 1988, Michael Deering proposed an idea in a paper he wrote titled *The triangle processor and normal vector shader: a VLSI system for high performance graphics* that detailed the initial idea for deferred rendering, though never actually naming the technique “deferred rendering.” [1] Though the idea was detailed in 1988, it took almost two decades for the idea of deferred rendering to become a reality. In 2004, Nvidia gave a talk at that year’s Game Development Conference titled “6800 League’s Under the Sea: Deferred Shading.” With that generation’s advancement in three dimensional computer graphics, it really brought the concept of deferred rendering back to the forefront. A year later, the developer’s behind the game, S.T.A.L.K.E.R., implemented a deferred renderer into their game. They released a paper detailing the experience they got from creating the deferred renderer for the game in GPU Gems 2 and compared it to the results they would have gotten from a forward renderer. [2]

|  |  |
| --- | --- |
| Figure 1: S.T.A.L.K.E.R Using Forward Rendering [2] | Figure 2: S.T.A.L.K.E.R. Using Deferred Rendering [2] |

## Compared to Forward Rendering

Before we can compare deferred rendering against forward rendering, we will first talk about the forward rendering pipeline. Forward rendering is a process that handles all the inputs for a given pixel being rendered to the screen at once. This includes calculating lighting information for each pixel. Unlike in deferred rendering, the lighting for that pixel is calculated in the same pass for that pixel. This can be somewhat taxing the more lights there are in the scene since it adds more calculations needed for that pixel before the next one can be rendered.

An example of an engine that used both a forward renderer and a deferred renderer would be the Leadwerks Engine. [3] The developers behind the Leadwerks Engine implemented a forward renderer as a baseline before implementing their deferred renderer. Comparing the two, the developers found that the deferred renderer was simpler to control. The largest advantage of doing this was that the lighting calculations are done in screen space rather than in world space like forward rendering. This is an advantage because it allows for fewer calculations on the GPU using a deferred renderer than a forward renderer. The implementation that the Leadwerks Engine utilizes is a three pass system that extends from the traditional three pass system. The significant difference the Leadwerks Engine makes is in its G-Buffer pass. Instead of the G-Buffer outputting three textures to send to the lighting pass, the Leadwerks Engine creates a fourth texture for specular lighting factors. This fourth map is generated from specular information from the meshes rendered in the first pass. This is advantageous to both the designers and developers since it allows for a degree of control in managing the lighting. It also makes the lighting calculations easier to perform since specular highlighting does not need to be calculated on the fly by the lighting pass. An extension of the fourth buffer allowed for the developers to render bloom or other effects without altering the engine code.

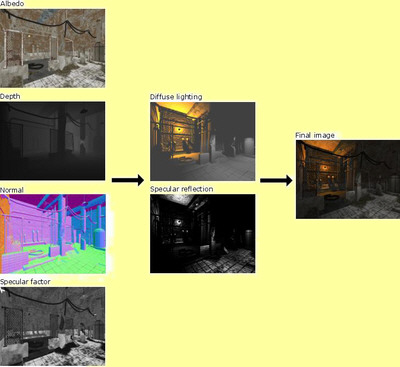


Figure 3: Deferred Rendering Pipeline in Leadwerks Engine 2.0 [3]

## Strengths and Weaknesses

Building a deferred renderer can have many advantages but also many drawbacks. Choosing to implement a deferred renderer is largely dependent on what type of environments games implementing the renderer will need to produce. One of the strengths of a deferred renderer is that it is flexible enough of a pipeline to use larger number of lights per scene than a standard deferred renderer. As mentioned above with the Leadwerks Engine, lighting calculations are done in screen space in the second pass of the deferred rendering pipeline, so there are fewer calculations that the GPU must perform before completing the lighting pass. This aspect of a deferred renderer really shines in games with extensive indoor environments due to them traditionally having more lights in the scene to worry about. As more lights are added to scenes, the value of a deferred renderer goes up.

The biggest drawback to having lighting calculations done on the final screen rendering is that it makes a couple techniques rather difficult to pull off. The most noticeable of these techniques that suffer from a deferred rendering pipeline is alpha blending. The reason for this is once the deferred rendering gets past its G-Buffer pass, it has no knowledge of object specific material and texture information. It only reads the buffer information for the final scene. There have been techniques to combat this issue but are often very taxing on the system due to the extra passes or buffer information stored per render cycle. One such technique was developed by Louis Bavoli and Kevin Myers of Nvidia using a technique called Stencil Routed K-Buffering. [4] Bavoli and Myers define the K-Buffer as the data storage of stencil tested depth fragment information for a pixel. The idea behind using a K-Buffer is to re-purpose a multi-sample render target that uses the samples to implement the K-Buffer. This K-Buffer stores up to 8 fragments per pixel which is combined through a geometry pass and alpha blended in a sliding window fashion. While the technique is successful, it is a fairly taxing process. This comes from having to sort the fragments before sending them off to the GPU due to the order the rasterizer receives them.

Another disadvantage of deferred rendering is that it is difficult to do hardware anti-aliasing. This is due to the geometry being separated from the geometric stage and therefor the antialiasing does not produce correct results. There are ways to get around this issue. Antialiasing can be applied at the first pass in the G-Buffer but it is often the case that it is not until the lighting pass is completed where aliasing is needed in the scene. Another common work-around to this is to perform an edge detection algorithm in the final pass and blur the edges slightly.

# Component Based Engine Model

A strong engine model that powers games is just as important as the game itself sometimes. When developers are looking to create games, they either build their own in-house engine or utilize a licensed engine from another source. One such model for these engines is the component based engine model. A component based engine (CBE) is a structured series of modules that encapsulate functions and data to a single entity. [5] In essence this means that a game using a component based engine can plug-and-play different modules onto a game object within the game world and have that module’s functionality acted upon the game object. Game objects are arbitrary bins that the components can attach themselves to. Once components are attached to a game object, they are registered with their respective tasks or directors that update the core functionality of the component. In component based engines, few core systems exist in order to create a baseline for the engine. From there, modules extend from the core system and attach themselves to the engine’s update cycle.

One of the most common examples of a component based architecture is the Unity Engine. [6] The Unity Engine is development suite that allows developers of all skill levels to create demos and games rather quickly with its plug and play style of design. The engine itself is a component system where the components are modules that users can attach to already existing objects or brand new ones.

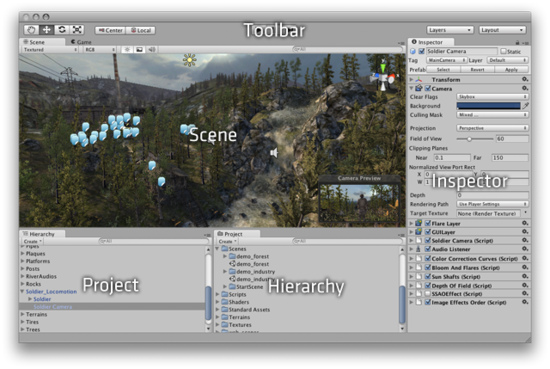


Figure 4: Unity3D IDE Interface [6]

## Compared to Hierarchal Based Engine Model

While component based architectures offer a level of modularity, they are not commonly found in engine designs. The most common engine design and thought process of programming is the hierarchal based model. The hierarchal based engine model is a tried and true design format that calls for developers to create a core that can be extended down through the various systems needed in the engine. This network of systems forms a dependency tree within itself that can be rather large depending on the robustness of the engine design. Often times, the layers of inheritance of these systems are long and narrow in design. This differs in component based models where the layers of inheritance are often shallow and wide. When a hierarchal system goes through its update cycle, it usually has a central system to process all the engine’s required processes. As mentioned before, in component based systems, the registered components have their own local processor.

## Strengths and Weaknesses

One of the largest strengths a component based model has under its belt is the ability to plug-and-play modules onto gameplay objects. This is especially effective if developers look to keep the engine software separate from their game development. This allows for quicker implementation of gameplay elements in a development cycle since no functions or data has to be overwritten or extended on the gameplay side of development. Alongside the plug-and-play idea for game objects and their components, this concept also applies to the systems implemented in the engine itself. This means that if a whole system, say artificial intelligence, was taken out of the engine, the engine would be able to run flawlessly as if the AI was not implemented at all. This is a very powerful notion since it means an engine can be as robust or simple as the developer sees fit for the game.

The drawback to this type of system is that should developers require specific functionality out of a component, it is rather difficult to add this functionality into the game. While the engine structure preaches a very modular design concept and the layers of inheritance are so shallow, the system can have a hard time detecting the difference between some of the components. This is due to the abstractness that the component system preaches. Another drawback to the component system is that while it preaches modularity, it has little to no concept of exactly what its system structure could be. Depending on what components a game would require from the engine could differ from what another game needs from the very same engine. Because of this level of abstractness, its update process turns a blind eye to the specifics of what modules are setup within the system.

## Singularity Engine

The Singularity engine was the engine inspired by the component based engine found in Unity. The author and his teammates designed a system utilized the component based engine model on top of a multi-threaded system that preached the idea of “no time being wasted.” Because of this, the engine ran very efficiently and fast and made for the development of their capstone game very quick. The Singularity engine created a tasking system that would be able to dynamically size its performance up and down based on the hardware it was running on. This tasking system would then create a series of threads that would spread the work load of the game evenly in order to maintain an efficient update and render cycle.

A few standards were made for the engine to run smoothly. The first was that of all the modules that the first iteration of the engine had available, the engine would require its “Core” and “Common” modules to be included as a baseline. Within the Core module was the tasking system with did the brunt of the work for the engine. The Common module housed the math and timing systems used for the update loops in the Core. The other core system, though not completely required for the engine to run is the graphics module. The graphics module is comprised of renderer components that are registered to the graphics task. This task handles all renderable objects that the system has registered. This includes any modules that are extensions of the graphics module such as animations, particles, and graphical user interface modules.

For the purpose of the author’s capstone game, *Trigger Happy*, the following modules were used:

* Animations – Houses animation processing and passes rendering components to the graphics module.
* Audio – Handles all audio playback and filtering
* Common – Stores timing information for the tasking system. Not an actual component that can attach to game objects
* Content – Handles all content loading for the engine
* Core – The work horse module. Not an actual component that can attach to game objects
* Graphics – Handles all rendering for the engine
* Gui – Extends from graphics to handle all graphic user interface options for menus and in-game HUD’s
* Inputs – Handles all user input in the game.
* Networking – Takes care of networking games together on a multi-cast based network architecture
* Particles – Extends from graphics to handle all special rendering effects
* Physics – Handles all collision parsing and physics response info for game objects
* Scripting – Adds Lua scripting functionality to the engine. Allows for other modules within the Singularity engine to be template and parsed through Lua

# Analysis of Deferred Rendering in Singularity Engine

For the research implementation of the capstone project, the author attempted to create a deferred renderer that made use of more than one material style in its rendering process. This proved to be a challenge given the nature of the Singularity engine’s component design. Two things had to be accounted for in the implementation of the deferred renderer; maintaining the overall design of the Singularity engine’s architecture, and making a renderer that was modular enough to handle the plug-and-play design of a component based model.

In order to understand the author’s research approach using the Singularity engine, we must detail the Singularity engine’s graphics module. There are three objects to keep track of; Camera, HardwareDevice, and DrawingContext. The Camera object is what handles updating the viewport of the application. It is inside the Camera object where the clearing of the back buffer on the device occurs. Once that initial clear is triggered, the graphics module’s task processor handles the rendering of all game objects to the screen. The Hardware Device object directly links to the graphics device and creates the application viewport. The second is the DrawingContext object is the root context that handles the base level render calls. It is from this object that the author’s deferred renderer implementation inherits from. The reason for this abstract context object is so that both deferred and forward rendering implementations can be implemented should the developer choose to. In order to be able to handle any extensions of the graphics module that the engine might have added, a layering system was added to the context’s system. This layer would allow for the separation of the various renderer components so that they would draw in the correct order. The layer system would be clamped from 0 to 100 so that An example of this can be found in the author’s capstone game, *Trigger Happy,* where they set static mesh objects registered directly to the main graphics module are set to layer 0 and objects registered to the GUI module are set to layer 100. The layer system reads in as objects with a higher layer are closer to the camera’s position so they take precedence in overlapping objects.

The implementation that the author chose to implement was the standard three pass model found in most deferred renderers to date. The G-Buffer pass would hold the diffuse, normal, and depth buffer information in three separate render target textures to be stored for later. The lighting pass would take the normal and depth buffers from the G-Buffer and add in the light calculations to create the lighting map for the final combine pass. The lighting system implemented in the Singularity engine allowed for lights to be added and deleted on the fly in run-time. Because of this, one every draw loop, the lighting information would be grouped together and passed in as one set of data to the lighting shader for calculation. The final pass would take the diffuse and light maps created in earlier passes and create the final combine map to be rendered to the screen.

Ultimately, this is where the implementation would stall. Upon further study of the Singularity engine’s structure, it became clear to the author that this implementation would not work beyond rendering static meshes that are registered directly to Singularity’s graphics module. The deferred renderer implementation immediately breaks when there is a module present that extends from the graphics module. The reason for this is that deferred rendering is meant for 3D applications only. Renderables such as a GUI are not done in 3D space so information such as lighting is irrelevant for it. The idea was for the deferred renderer to use the layer system within the engine to determine which material process the renderer would use. This suffers from the same problem of module knowledge as the rest of the engine. We would have to specifically state which layer uses which material process which in turn requires intimate knowledge of the module layout for the engine. Ultimately this does not keep with the plug-and-play architecture the Singularity engine looks to keep.

# Conclusion

In the end, the biggest feature of the Singularity engine was the deferred renderer’s downfall. As a backup, a buffered forward renderer was designed for the Singularity engine and served the author and his development team admirably. In order for a deferred renderer to work within the Singularity engine, the structure of the engine itself would have to be modified in such a way that object recognition was universal across the engine. The deferred renderer would need to access this knowledge of the engine’s structure and be able to dynamically filter the different renderable object types. This would be need to be done in such a way so that 3D objects rendered into the world utilize the deferred process where other objects such as GUI objects would not.

# References

[1] Wikipedia contributors 2010. “Deferred Shading.” Wikipedia, the Free Encyclopedia. Available online at <http://en.wikipedia.org/wiki/Deferred_Rendering>

[2] Shishkovstow, Oles. 2005. “Deferred Shading in S.T.A.L.K.E.R.” In GPU Gems 2, edited by Matt Pharr, pp 143-166. Addison-Wesley. Available online at <http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html>

[3] Josh Klint. 2008 “Deferred Rendering in Leadwerks Engine. “<http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf>

[4] Louis Bavoli, Kevin Myers 2007, “Deferred Rendering Using a Stencil Routed K-Buffer.” ShaderX 6, pp 189-198.

[5] Wikipedia Contributors 2010. “Component Based Software Engineering. “ Wikipedia, the Free Encyclopedia. Available online at <http://en.wikipedia.org/wiki/Software_componentry>

[6] Unity Development Tool <http://unity3d.com/unity/>