15-462 Project 4: Deferred Rendering

Release Date: Thursday, March 26th, 2015

Due Date: Tuesday, April 28th, 2015

1 Overview

In this project you will design and build a real-time deferred renderer for static scenes with dynamic lighting. Part of the purpose of this assignment is for you to gain experience in programmable shading with a modern graphics API, so most of the design and code will be your own. However, we require that your renderer support the provided scenes and lighting system using a deferred rendering approach. The rest of this document outlines the base requirements, the provided code, and a suggested overall design for your renderer. This is not meant to be an authoritative design specification, but rather an outline to get you started. Your goal is to build a system that produces high-quality visuals, and we encourage you to use the project as a base for exploring advanced techniques in the future. Extra credit may be awarded for renderers which go beyond the scope of this document in quality of rendering or additional features.

2 Goals and Deliverables

Your goal in this project is to build an executable which reads scene files (as text) and renders the scene to the screen from the perspective of a user-controlled camera. Unlike earlier projects, these may be large scenes containing dynamic lights, and your renderer will be expected to handle a full lighting model including shadows.

Your deliverables include:

Source Code You will be implementing a rendering application in C++ and OpenGL. Your final submission should include all source files, including shaders, used in your project.

Documentation In addition to in-code documentation, you will create a written design document and development report **pdf**. See Section 6 for writeup requirements.

Grading will be based primarily on the quality of your results and code (60%) with up to 15% extra credit for outstanding achievements. Your written report will make up 35% of your grade, with the remaining 5% based on project checkpoints.

3 Code and Programming Requirements

3.1 Provided Code

To help you get started, we provide minimal starter code which handles window management, initializing an OpenGL context, and loading scenes from text files (you will need to perform additional processing of scenes before rendering - see Section 4). You can download the code from afs:

/afs/cs.cmu.edu/academic/class/15462-s15/www/project/p4/deferred/code.zip

There is also a copy in Felipe's Github account, which you can **fork** if you would like to maintain your code on Github:

`http://github.com/fgomezfr/462deferred`

This will allow you receive any updates or bugfixes to the starter code, so it is encouraged that you use git (or an alternative source control system of your preference) for this project. Either way, you are responsible for your code; *no extensions or grading relaxations will be given for 'lost' code.*

3.2 Usage and Requirements

You are not strictly required to use the starter code we provide but it may save you some time. Please take a moment to review the provided code, and in particular follow the links in the `README` for instructions on installing SFML, a cross-platform multimedia library used for window management, keyboard input, and handling OpenGL. The starter code is designed to compile and run on any platform provided SFML is properly installed; regardless of whether you use SFML in your implementation, it is expected that your code will also build and run on any capable machine.

To achieve this, your renderer **MUST** be written in OpenGL. Any version at least 2.0 is fine provided it meets your needs, but you may **NOT** use the fixed-function pipeline - you must implement all shading work (transformations and lighting) in custom GLSL. By default, the starter code targets OpenGL 3.0 - you can change this by setting the appropriate members of the `sf::ContextSettings` variable in `main()`. Check your graphics card specifications to see which versions are supported by your hardware; for grading, we have access to versions up through **4.3** on a Windows machine.

In addition to SFML, the starter code includes GLM, a header-only math library which implements many common graphics operations (e.g. building a perspective matrix). GLM's vector and matrix libraries should feel similar to the scalar implementations you saw in previous projects, but can compile to SIMD-vectorized implementations for better performance. We encourage you to use GLM in your project, as it provides many of the operations you are used to from **glu** and the **_462** math libraries but is not bound to the OpenGL matrix stack; see the online documentation for more details.

3.3 Executable Requirements

Your compiled code should:

1. Load a scene file (from a path provided as the final command line argument). Most of this is handled for you, but not all necessary processing is performed. See the next section for a brief overview of the scene files; your code should correctly handle scenes in the provided format, but you may extend the scene specification as you see fit (see section 4).

2. Render the scene to a window with a user-controlled camera. Controls are up to you (please document them), but we recommend using WASD+QE for translation and arrow-keys or mouse for rotations. The camera should be able to fly freely through the scene at a controlled speed.[1]

3. Maintain an interactive framerate (preferably $\geq$30 Hz) at reasonably high resolution (you should of course aim for 1920×1080 or higher, but lower resolutions are acceptable). Performance on complex scenes is difficult to predict, but with good techniques the lab machines (with GTX 480-780 GPU's) should be able to produce high framerates under directional lights. One open challenge in this project is to efficiently render a scene with many moving point lights. Rendering scenes with hundreds or thousands of lights is not an easy problem, so we aren't setting any performance expectations up front - make this a personal challenge!

4 Scene Files

4.1 Scene Description

Your renderer will load scenes in the custom `.scene` format (which is really just a text file). This can yield slow load times as multiple files need to be parsed and data processed into a form fit for use in rendering, but allows easy modification of scenes and sharing of large objects between scenes. Please see the file specification included with the scenes for details on the file format. As a general overview, each scene consists of:

[1]Restrictions such as camera collision with objects or a realistic 'character' camera as in an FPS game may be accepted for extra credit.

Models A collection of static models with associated world position, orientation, and scale. Transformations are defined in the scene file, while models are defined by `.obj` files using relative paths. The provided code reads the data from each `.obj` file to a similar in-memory object form. While convenient as a universal object definition, you will find that `.obj` data itself is not very amenable to efficient rendering, so you will have to organize the loaded data (mesh vertices, faces, and materials) into a convenient form. Keep in mind that `.obj` files can be very large, and might contain more than one scene 'object' - in fact, some scenes are contained entirely in a single file! Properly organizing data is an important technique for efficiency, so start with simple scenes and iterate frequently on your mesh-handling code.

Directional Light The entire scene is uniformly lit (not counting shadows) by a single directional light. Your early versions might render using only this light for simplicity; the directional light is static and does not change during program execution.[2]

Spot Lights A collection of spot lights in the typical OpenGL form (position, direction, cutoff angle, and falloff exponent). Spot lights are also static, but you can earn extra credit by making them animated (for instance, using SLERP to 'swivel' between two orientations).[3]

Point Lights A collection of point lights with position, a cutoff radius, and a velocity parameter in the range $[0, 1]$. If the velocity is zero, the light is static and does not move; otherwise, it should move around the scene, preferably randomly, using the velocity as a parameter. The exact animation is up to you, but we require that your renderer support moving point lights.[4]

Materials used in the scenes are properties of the models. You are not required to support the full range of materials that can be specified in `.objmtl` form, but all models will have a diffuse color and your renderer must support some form of specular highlights and indirect light. Most models will have typical (K_d, K_s, K_a) materials so a Blinn-Phong shading system is acceptable, but we encourage you to investigate more advanced lighting systems and create scenes that demonstrate your additional features, as long as the needed values can be specified in a `.objmtl` file.

Some scenes may contain alpha-blended materials; these require special handling (especially in a deferred system), so we won't require you to alpha-blend them, but this will be the first thing we look at for extra credit.

[2]For extra credit: Extend the scene specification to allow animated directional light, and create scenes that simulate something reasonable, e.g. the sun's motion overhead. In your writeup, discuss the potential advantages and drawbacks of requiring the light to be static.

[3]This is a performance challenge as well as a mathematical one: Animated spot lights require shadows to be re-generated every frame!

[4]Try to create interesting effects with your animation, if you have time. For extra credit, you could prevent light intersection with objects, implement simple flocking algorithms, or add more parameters for interesting motion paths.

4.2 Provided Scenes

A small collection of models and scene files is available on AFS:

`/afs/cs.cmu.edu/academic/class/15462-s15/www/project/p4/deferred/Scenes.zip`

You should fork this repository and clone it to your local machine. Since some object files can be rather large, we also maintain a copy on AFS at

`/afs/cs.cmu.edu/academic/class/15462-s15/projects/scenes/`

This a large file, so you will want to download and extract the scenes to your working machine. If you are working on GHC linux machines, you can extract files to `/tmp` where they will be accessible to all users.

You can and should modify and add to the scene files to experiment with your renderer's capabilities. In particular, we have not provided scenes with spot or point lights; lighting should be the focus of exploration and customization in your design. You have artistic freedom to adjust all lighting in the scenes, or create new scenes to showcase your lighting system. If engineering is your focus, you might try to render a scene with thousands of randomly-placed lights - this won't be easy!

There are many additional resources, including free models and materials, available online. Feel free to create your own scenes; if you create something interesting and would like to share it with the class, email course staff and we can add it to the download package.[5]

4.3 Extending the Scene Specification

To support additional features in your renderer, you may freely extend the `.scene` file specification as you see fit, provided your extensions are in fact 'extensions' and don't conflict with any existing requirements. Please document any such extensions in your write-up, and ensure that your renderer still works with the provided scenes.

5 Deferred Rendering

5.1 Rendering Basics

By now you should be familiar with rendering via perspective transform and rasterization as implemented by the modern graphics pipeline. You have also implemented a raytracer, which takes a different approach to generating images by casting rays backwards from the camera into the scene. Both techniques solve the same fundamental problems in rendering:

[5]Be professional when downloading content; give credit where it is due, and include the copyright or license if it has one. Also, many of the scenes we provide are well-known standard scenes, licensed by the original creators. They are provided *solely* for use in this project; do not duplicate and distribute these files or use them for commercial works.

Visibility - finding the first non-occluded surface in each viewing direction

Shading - computing the color at each visible surface point

Raytracing handles the visibility problem by intersecting rays against the scene to find the closest intersection point, and then performs shading calculations by casting recursive rays. Traditional 'real-time' rendering instead uses the *depth buffer* to perform visibility testing: Each object is projected to the screen, and for each pixel sample, the distance to the surface is recorded. Shading is then performed independently for each object, and the visible color recorded only for the closest known surfaces.

Hardware-accelerated rasterization and depth-buffering provide a fast means to solve the visibility problem, so in a modern renderer much of the work is instead spent on shading. This leads to performance problems in scenes with high depth complexity and expensive lighting computations. For example, in the traditional rendering pipeline - also called 'forward' rendering - a surface which passes the depth test must be shaded and the color saved to the frame buffer; but all of the work done to shade the surface may be wasted if a *closer* surface is later found to cover the same region of the screen!

This problem is exacerbated for scenes with many lights, since the cost to shade a single pixel can be very high. Another issue is that iterating over a long list of lights in a shader for each visible pixel can be very expensive; with many lights in the scene, we would like to efficiently decide which lights are necessary and only apply them to affected regions of *visible* surfaces.

Both of these problems can be solved in a 'forward' rendering system, but many of these solutions have additional drawbacks. For instance, you might use a 'depth pre-pass' in which geometry is projected (without any shading) to fill the depth buffer, and then re-rendered so that shading is applied only to the true closest surface. This avoids wasted shading work, but requires transforming and rasterizing all objects twice! For lights, you can often predetermine which lights affect which objects and create short lists for rendering, but this has problems with animated scenes[6]. These and other issues have led to the rise of a family of techniques called 'deferred' rendering, which we would like you to explore in your project.

5.2 Simple Deferred Rendering

The core concept of deferred rendering is simple: rather than perform expensive shading computations up-front, we can 'defer' them until after visible surface determination is complete. Rather than recording the final color in the frame buffer, we record each surface's *material* properties in a 'geometry' buffer or 'G-buffer.' After all objects have been projected to the screen, we can perform a full-screen pass over the geometry buffer to compute the shaded color.

Deferred rendering thus solves the wasted-shading problem by performing a minimal amount of shading work up-front (getting material properties) and

[6]There are also other performance issues related to GPU parallelism; take 15-418 and talk to your TA for more info.

waiting to perform any work that might be overwritten, at the cost of additional memory to save the material values. You can think of this as the classic 'recompute vs. store' problem - a depth pre-pass solves the same problem at the cost of *recomputing* world positions to extract material parameters, while deferred rendering *stores* the material values to avoid re-rendering the geometry.

Deferred rendering also opens up new solutions to the lighting problem, because the depth and geometry buffers contain the surface data needed for all and only the visible surfaces! Rather than testing each surface against all lights, one simple technique is to apply each *light* to the visible surfaces, and *accumulate* the resulting color in the frame buffer (often called a *light accumulation buffer*). For example, if each light source only affects surfaces within a specific volume, we can render that volume to the screen as geometry, and only apply the light for pixels within the projected area.

5.3 What You Should Implement

There are many possible techniques for deferred rendering, including tile-based and hierarchical approaches for quickly culling and rendering thousands of point lights, and a full discussion of them is beyond the scope of this document, so we **strongly** encourage you to seek additional resources to guide you in design and development of your renderer; many example techniques can be found online. As a general outline, we suggest the following form for your renderer:

Buffers You will need to maintain (at minimum) a *depth* buffer, a *g*-buffer storing 1) surface normals 2) diffuse albedo and 3) specular power and/or a material ID, and a *light accumulation* buffer.

First Pass Project all geometry to the screen, recording the material properties for the closest surface in your g-buffer.

Directional Light Pass Perform a full-screen pass (you can do this with a compute shader or by rendering a rectangle which covers the screen) and apply directional light to all surfaces, saving the color in your accumulation buffer.

Spot Light Pass For each spot light, render a cone containing the spot light's volume of influence. For each pixel covered by the cone, compute the light contribution from the spot light at the stored surface *if* the surface lies within the cone, and add the result to your accumulation buffer.

Point Light Pass For each point light, render a screen-space circle covering the light's projected sphere-of-influence. Accumulate the light contribution for each covered sample whose depth value lies within the light's sphere-of-influence.[7]

[7] This is only one suggested technique; you can also just render squares and perform a point-in-sphere test, or use a tile-based compute scheme.

The outline above describes a basic deferred rendering scheme, and will probably not be sufficient for very complex scenes. In particular, your renderer will likely benefit from some sort of *visibility culling* technique to avoid rendering geometry that lies outside the view frustum, and you might consider an acceleration structure for scenes with many point lights.

5.4 Lighting and Shadows

For this project, the design of the renderer and lighting system is up to you, except that you must implement a deferred system like the one outlined above. We expect that most students will implement a simple shading model such as a Blinn-Phong model, and this should be sufficient for most scenes given the format of provided `.obj` data. Another important topic in current rendering systems is proper handling of *indirect* or *global* illumination, as opposed to the simple 'ambient' term in a Blinn-Phong system. However, implementing such a system is a large project typically involving pre-computed radiance transfer data, and requires knowledge of an existing renderer and scene structure. For this project, we won't require anything more involved than a simple 'ambient' term using the intensity of each light and the provided K_a values for each material.

We *do* require that your renderer support *some* reasonable ambient model, and combine both direct lighting and specular highlights in rendered images. Additionally, we require that you render accurate shadows for directional lights and spot lights (but not point lights)[8]. There is a wide space of techniques for rendering shadows, but you should probably stick to a simple *shadow mapping* technique. Part of your grade will be based on the quality of your shadows - are they aliased or pixelated, do you approximate soft shadows, and if you implement dynamic lights how well can your shadows keep up. Shadow quality is one area where you can potentially invest quite a lot of time, and we may reward the best solutions with extra credit!

Extra credit may also be awarded for additional features such as environment-mapped or screen-space reflections, ambient occlusion, tessellated geometry, or advanced materials (e.g. bump-mapped surfaces). Please document any such features you implement, provide scenes to demonstrate your results, and be prepared to showcase your renderer for the class!

6 Project Checkpoints

Since this is a longer project with a substantial design requirement, we have set a basic timeline for you to follow. You will submit screenshots and a brief written statement for two checkpoints, so that we can tell if you are on track. You earn full credit for the checkpoints just by submitting, so your grade will not be hindered if you fall behind; however, this will let us know if you are (or aren't)

[8]Point light shadows can be done with cubemaps, but this can be very slow especially with animated lights, so shadows from point lights can be extra credit.

making progress, so we can help and set expectations for the class. All files for checkpoints should be submitted to your `/15462-s15-users/<andrewid>/p4` directory on AFS; submit screenshots in a standard image format, and written comments in a text file named *<andrewid>_<checkpoint#>.txt* Below is the timeline you should aim to follow if you want to complete the basic requirements; you may want to set earlier deadlines for yourself if you are not familiar with OpenGL or C++ or intend to explore extra-credit features:

Week 1 (3/27 - 4/02) Generate meshes from obj models, implement basic camera transforms.

Week 2 (4/03 - 4/09) Render depth-buffered images, build material buffers and begin implementing directional light.

Checkpoint 1 - Due 4/09 11:59pm Submit a rendered image of your favorite scene showing proper surface occlusion (i.e. render a depth buffer to the screen). Write a brief description of your G-buffer design - what values do you plan to store, and how are they organized in buffers?

Week 3 (4/10 - 4/16) Apply directional light; add shadows, and begin rendering perspective shadows for spot lights.

Checkpoint 2 (Optional) - Due 4/21 11:59pm Submit a screenshot showing color and shadows in your favorite scene. Write a brief description of your approach to applying spot and point lights.

Week 4 (4/17 - 4/23) Add spot lights (with shadows) and point lights to your rendering. Write some test scenes for your animated lights.

Final Weekend (4/24 - 4/28) Finish debugging; complete your writeup! Breathe!

This is an open-ended project, and there is a lot of room to go beyond these basics and build and impressive renderer. However, your time in this class is limited, and you are still responsible for learning the lecture material. Let us know early if you are falling behind; if you are comfortable with OpenGL and can implement lighting quickly, try adding other features to your renderer - physically-simulated animations and indirect lighting come to mind!

7 Writeup Requirements

Since this is a design project, a larger part of your grade than usual is allocated to documentation. Along with your code, you will submit a pdf document describing your implementation goals, what you accomplished, and your experiences along the way. In your writeup you should:

1. Document usage of your renderer - camera controls, command line arguments, and any additional parameters.

2. Describe the overall design of your renderer - explain the pipeline process of rendering a frame, and how materials and lights are handled by your system. Don't go into too much detail here; we want a high-level overview of how you renderer works.

3. Briefly describe how you handle scene geometry - did you make any optimizations in organizing scene data? Do you apply any culling techniques or hierarchical structures?

4. Explain your shadow-mapping implementation; what were your goals, and how did you achieve them? In particular, analyze the quality of your shadows (e.g. soft shadowing and aliasing). If your results are not always very pleasing, explain why, and suggest a change that could improve your shadow quality. Otherwise, explain what additions or changes you made to the basic shadow-map algorithm. Did you explore filtering, cascade shadows, or some other technique? What worked well for you, and what would you like to try if you had more time? Include images showing good and bad cases of shadows in your renderer.

5. Describe how you handle point lights, and assess the capabilities of your renderer. Did you try to handle many lights, and if so how successful were you? Include a few screenshots of well-lit scenes.

6. Briefly discuss the performance of your renderer; if you weren't able to achieve good framerates on the larger scenes, try to identify why. If you were successful, explain what additions you made to handle complex lighting environments, such as many moving point lights. How far did you try to push your renderer, and what did you do to accomplish that?

7. If you made any new features to your renderer or the scene format, describe your additions in detail. What effect were you trying to achieve, and how did you go about doing it?

Your writeup should both describe what you've created and explain the process you took to achieve your goals. If you encountered problems, explain how you solved them; if your design changed, explain why. We want to see what goals you set for yourself, how you pursued them, and what you accomplished. Since the design of your lighting system is up to you, the performance capabilities and quality of rendering are also yours to decide. Set a bar for yourself, and explain what you did to reach it. We will judge your achievements in comparison to what you've learned in the class, but a large part of the grade comes from your writeup. This is an opportunity to justify what you did and didn't accomplish in this project, and show what you learned in the process.

8 Submission

Submit your source code (including CMake build system), writeup, and any interesting scene files you would like us to consider while grading to your `/p4` handin directory on AFS.[9] In addition to your submission from the project checkpoint, we will be looking for three items:

A `/src` folder containing all of your code. This should contain `CMakeLists.txt` files allowing us to generate build files on any platform. If you made significant changes (besides adding C++ classes and headers), include a short README describing your changes and what platforms you have tested on. *We would like your code to work on any platform, but dealing with CMake is not a focus of this project. We will try to grade your code on the same platform if necessary, but we can't promise an equal environment. You might include links to video recordings of your renderer at work as a backup if you encounter problems building with CMake.*

<andrewid>.pdf Your writeup, including embedded screenshots of your renderings.

A `/scenes` folder containing some example `.scene` files showcasing lighting in your scenes. These should be intuitively named and work in the same directory structure as the Scenes handout. Please submit only text files; if you created custom scenes and would like to submit a full scene structure with custom .obj models, please contact course staff.

[9] If you are using Github, you can also provide us a link to clone your repo; talk to Felipe before the deadline.