CS 283: Assignment 3 — Real-Time / Image-Based Rendering

Ravi Ramamoorthi

1 Introduction

In this assignment, you will implement some of the modern techniques for real-time and/or image-based rendering. To account for varied interests of students, I provide three choices for this assignment. You need only pick one of the choices, based on your interests and expertise. Since many of these are close to the research frontier, I am providing as much guidance as possible, but you will need to be proactive in looking up the original literature and other sources such as online tutorials. No source/skeleton code is available. If you find online source code, you are welcome to use it as a reference, or to use utility functions, but obviously not as a core part of the assignment. If in doubt, please ask the instructor for clarification.

This assignment should be completed in groups of two. Both partners will receive the same grade. If you absolutely want to work alone, you may do so, but similar requirements will apply. It will be turned in by creating a website. Please note that since the emphasis is on real-time, *you must provide a link to a video of your system running interactively, using a widely readable codec. Many demos don't actually run on our systems, so we have no good way of grading without a video.* Future editions of the course may provide more support, but the point is also to get you to work independently and communicate your results, given that this is an advanced graduate course.

Choice of Assignment: There are three choices provided for this assignment, that correspond to three important current research directions in rendering and appearance. You only need to do one of them (they are listed in separate sections below).

- *Precomputed Radiance Transfer System:* The technique of precomputed radiance transfer is a popular method over the last decade for rendering and relighting. By implementing a project along these lines, you will develop a basic background in real-time rendering in general, and in this new approach to relighting. I would suggest you start by implementing only the simplest basic system, such as Ng et al. 03, and if time permits, add to it. Students working on this project will have both mathematical and systems skills, with an interest in learning state of the art real-time rendering techniques.
- *Basic High Quality Rendering:* Given the power of current graphics cards and computers, a number of high quality visual effects can be included in the real-time rendering domain. This assignment asks you to explore many of these concepts. At a minimum, you should use GLSL shaders to implement basic shadow and environment mapping. You should also include some other visual effects. An ambitious goal would be include more general BRDFs, using something like Cabral et al.'s reflection-space image-based rendering. But even if you don't do that, create some interesting visual effects and a really nice demo of high quality real-time rendering. Students working on this project will have strengths in real-time programming using OpenGL (or DirectX), and an interest in fast hardware rendering.
- *Image-Based Rendering:* A popular new paradigm to rendering is to create new visual content from existing images by interpolating captured or synthetic data. While this assignment focuses on light field rendering, similar ideas with data-driven appearance are now widespread in many parts of the field. In this project, you will write a basic light field viewer, that will allow one to view real or synthetic data from various different angles. Note the parallels to the real-time rendering projects above: precomputed radiance transfer for relighting essentially allows you to look at an object under any illumination, while this project allows you to look at an object from different viewpoints. Both

techniques are geared to run in real-time. Students working on this project will have interest in object and material acquisition from real data, in addition to the usual systems capabilities.

Given the nature of these methods, being close to research in the area, your best source is often the original papers and online resources. You will have to do more groundwork on these projects, but it is also a good preparation for the state of the art research in the area. Choose which of the three projects above you want, based on your group's strengths and interests. They are described in more detail below.

2 Submitting the Assignment and Logistics

Since one of the goals of preparing for research is to also communicate clearly, the assignment must be turned in either by submitting a PDF by e-mail to the instructor (as you would a scientific paper), or by creating a website and sending a pointer to it. Since part of your results will involve interactive effects, we may also schedule individual demonstrations with groups to demo your system in action. Regardless, *you are required to also create a video(s) from screen-captures of your system, available from your website (as you would for a paper).*

Document your results properly, showing the various types of scenes your method can handle. Explain any interesting algorithmic aspects. It is likely you will need to create some test scenes; there are a number of resources available on the web for interesting geometric models and scenes that have been used in previous papers.

Failure to fully document the results of the program will likely lead to unhappiness from me. Be honest, and note what works and what doesn't; this will be the most fun, and make the assignment easiest to grade. Some part of the grade will be reserved for the clarity of the writeup.

If you make a website, do not modify it after the due date. In either case, please include links to the source code and executable, and again do not modify them after the due date.

Support: Feel free to contact me or the TA regarding questions. There is no code framework provided for this assignment. You are welcome (and encouraged) to look at online code sources and the original literature, and even the code as a reference, or to create utility functions. Obviously, you cannot copy code from elsewhere for a core part of the assignment though; ask me if there are any questions.

3 Precomputed Radiance Transfer and Relighting

Recently, a body of work known as precomputed radiance transfer, or precomputation-based relighting, has become a popular technique for real-time rendering in varying illumination, and recently also viewpoint for dynamic scenes. Good references are the seminal papers by Sloan, Kautz and Snyder in SIGGRAPH 02, and Ng, Ramamoorthi, Hanrahan in SIGGRAPH 03. A good historical source, that also surveys some of the previous work in image relighting, is a survey I recently wrote on precomputation-based rendering (available on both my website, and on the publisher's. I can also give you a spare copy of the book if you ask me quickly enough). In this context, I also encourage you to read some of the earliest work, such as Nimeroff et al. 94, "Efficient Re-Rendering of Naturally Illuminated Environments", EuroGraphics Workshop on Rendering, and Dorsey et al. 95, "Interactive Design of Complex Time-Depending Lighting", IEEE Computer Graphics and Applications, 1995.

The basic idea of precomputation-based methods is based on linearity of light transport. You precompute solutions for different light sources and then linearly combine them in some quick way. Ideally, you would implement a basic PRT system, at least along the lines of Ng et al. 03, and maybe some of the further work like Wang et al. 04 or Ng et al. 04. However, this is difficult in the short time available, so I stratify the various tasks below:

3.1 Basic Relighting from a few Point Lights

First, implement a very basic relighting system. Consider a scene with a fixed viewpoint (this is essentially the approach taken in Ng et al. 03). You can use your raytracer from the first assignment (or any other method, including a publicly available rendering system) to create images from a number of different properly chosen light source locations. For compatibility with later parts of the assignment, choose (say) 20 or so light source directions, which are directional lights, These could be uniformly distributed on the sphere, or chosen in such a way as to create interesting visual effects in your scene. Implement a simple visual system that allows you to change the intensities (and possibly colors) of these lights and re-renders the scene in real-time. This is done simply by linearly combining the source images, based on the light source intensities. This system provides a basic framework for interactive lighting design of images.

Similar to Ng et al. 03, you could also make the computations on vertices of a diffuse mesh, allowing you to visualize a Lambertian model in simple OpenGL, where you can also change viewpoint. Do this last part (vertex computation and meshes) before the other components, only if it is relatively straightforward.

3.2 Image-Based Lighting

The next step is to consider image-based environment lighting, as for example in Sloan et al. 02 and Ng et al. 03. Environment maps are now very common, and high-dynamic range lighting environments are available from a number of websites, like Paul Debevec's. You could also try painting your own. A common representation is a cubemap, where the sphere is projected onto 6 faces of a cube that are unwrapped.

Ng et al. 03 used cubemaps with resolutions up to $6 \times 64 \times 64$. However, the larger resolutions will make your system harder to build, and so I recommend starting small, with say resolutions of $6 \times 16 \times 16$, and perhaps images of size 256×256. Actually, if you implemented the vertex-based method above, you could try very small geometric meshes of about 10,000 triangles. Note that a cubemap resolution of $6 \times 16 \times 16$ is still 1536 images and the rendering precomputation could be slow. You may want to turn off global illumination in your ray tracer, and just do direct lighting, that could be much faster (it effectively reduces to visibility computations by tracing rays). If speed is still a concern, you could also try implementing the hardware rasterization approach in Ng et al. 03, where you go to each vertex or pixel, and rasterize the model to get visibility over the cubemap.

You now have a full light transport matrix. Assuming an image resolution of 256×256 and light resolution of 1536, this is about 100 million entities. This is still small enough that a floating point version could fit in main memory, but it quickly becomes intractable as you scale up cubemap and image resolution. You can see what rendering speeds you get without compression. It will likely not be real-time, but may be a few seconds for modern computers with low resolutions.

Note that you will likely want to implement something to manipulate the environment map lighting, most commonly by rotation, but you could also simply translate it along the cubemap. This will allow new lightings to be created, which shows the power of the method for relighting.

3.3 Wavelet and other Transforms

Your next task is to wavelet transform the transport matrix along its rows (i.e. along the lighting cubemap) as in Ng et al. 03. You can use the simple Haar wavelet for this purpose. You can also wavelet transform the lighting and choose only the first 100 or 200 terms, making the transport-lighting dot product tractable. See if you can achieve real-time framerates with this, and do some experimentation. Be careful to program the inner loop in a memory-coherent way for maximal efficiency. Having got this far, you can play with increasing resolutions, and using dithering/quantization, as well as truncating elements of the transport matrix to 0 as in Ng et al. 03.

If you complete the assignment so far, you may also want to try using low-order spherical harmonics instead, and compare wavelets and spherical harmonics. See if you can reproduce the types of graphs and comparisons made in Ng et al. 03. More ambitious groups may want to implement some of the methods in Sloan et al. 02, to produce real-time demos of the type one would want in video games. Even more ambitious extra credits are to do all-frequency lighting and view manipulation along the lines of Wang et al. In any event, if you go beyond the base requirements of the assignment, extra credit will be available.

Note that a key component of this assignment is the final demo of relighting with complex shadowing and/or global illumination effects. Noting which of the stages above you were able to complete, and the quality of the final demo will be important factors in grading.

3.4 Grading

The grading scheme is subject to change, but tentatively assigns 25 points for basic relighting, 30 for environment maps, and 20 for wavelets. The remaining points are 15 for documentation (including video) and scene selection, and 10 points for robustness and other issues.

4 Basic High Quality Rendering

The purpose of this project is to develop an understanding of the basics of modern high quality rendering. Towards this end, you will implement a basic program, likely using GLSL shaders, and including environment and shadow mapping. The instructions in this assignment are relatively brief; perhaps more detailed step-by-steps will be provided in a future instantiation of the course. The best source of more detailed instructions is probably online tutorials and user manuals (such as for GLSL). Note that you can probably find source code online to implement many of the core components of this (or any other) project. While you may look at that as a reference, do not directly copy it (when in doubt if a code segment is a core component or a utility aspect you can use, please ask).

4.1 GLSL Shaders

Modern graphics cards include the capability of developing vertex and fragment shaders, that provide control over various aspects of the graphics pipeline. They are therefore a key element to implement various high quality rendering effects. Shader development has undergone several iterations, including Cg and HLSL. However, they are now a core component of OpenGL 2.0 and 3.0 documented in the OpenGL red book with a GLSL (GL shading language) available. Your first task is to get basic GLSL programmable shaders working, based on the documentation in the red book and other online sources. Demonstrate a simple basic program that uses shaders to do something interesting, that may be more difficult or impossible in the standard OpenGL pipeline.

4.2 Shadow Mapping

Next, you will implement a common shadowing technique known as shadow mapping. The scene is first rendered from the light source, with depth values stored. There are special OpenGL $GL\_DEPTH\_COMPONENT$ specifications for these depths. This scene is then rendered from the camera, with depth values projected back to the light and compared to determine shadowing. The exact details of the technique can be found in many textbooks and online tutorials. The key issues are to set up the projection matrices correctly to project back to the light, and execute the right programs. You can implement this with GLSL shaders, or on graphics cards that support many of the SGI shadow extensions. Try to implement percentage-closer filtering for antialiasing (the basic idea dates back to Reeves Salesin Cook from SIGGRAPH 87). By applying this properly, you can also simulate penumbrae from soft shadows.

4.3 Environment Mapping

The other technique you need to implement is environment mapping. This gives the effects of mirror reflections from image-based lighting. For each point on the surface, you look up the surface normal and view direction, reflect the view about the normal, and use that to index into the lighting. The indexing is simply a texture lookup into the environment cube map. This should be relatively straightforward to program using GLSL shaders, although there are also techniques to do so using standard OpenGL. With shadow and environment mapping, you should be able to make nice images with complex lighting and shadows. Note that environment-mapped shadows are a challenging problem, so you may want to pick the brightest lights in the environment for shadow mapping.

4.4 Add-Ons

Besides the basic requirements above, you need to implement at least one add-on. The most natural example could be in environment mapping, where besides mirror reflections, you consider diffuse and Phong Shading.

You may also want to look at Heidrich et al. SIGGRAPH 99, "Realistic Hardware-Accelerated Shading and Lighting"for some ideas on interesting things you could do (many of the methods there could be implemented more simply than in the paper, using programmable shaders).

As described by Heidrich et al., and earlier in Miller and Hoffman, diffuse shading simply involves convolving your environment map with the Lambertian shading function (a cosine over the upper hemisphere). This diffuse map can then be looked up directly using the surface normal. For an easier procedural technique, read Ramamoorthi and Hanrahan, SIGGRAPH 01, "An Efficient Representation for Irradiance Environment Maps". For the specular, you instead convolve with the Phong filter, and look up based on the reflected direction. More ambitious projects may want to implement general BRDFs, using Cabral et al. 99, "Reflection Space Image Based Rendering" or Ramamoorthi and Hanrahan 02, "Frequency Space Environment Map Rendering".

With shadow maps, there is wide literature on various enhancements possible, including perspective shadow maps, adaptive shadow maps and so on. There are also various soft shadow mapping techniques that can "fake" shadows from an area light. If you are interested, talk to the instructor.

A number of other possibilities for add-ons also exist. In any event, the final goal is a high-quality real-time rendering demo that shows object(s) with a variety of visual effects being created.

4.5 Grading

The grading scheme is subject to change but tentatively has 25 points for basic real-time rendering (a solid video and demo is needed), and 20 points each for shadows, environment maps and add-ons. 15 points will be given for documentation and scene/video selection.

5 Image-Based Rendering: Light Field Viewer

Image-Based Rendering is an exciting new area of computer graphics. Instead of rendering synthetic images from "scratch", one uses acquired or rendered images as inputs, interpolating between them to render intermediate views. In this project, you will implement a viewer for one of the first image-based rendering techniques, Light Field Rendering by Levoy and Hanrahan in SIGGRAPH 96. This paper is very similar to The Lumigraph by Gortler et al., also in SIGGRAPH 96, but the former paper is conceptually cleaner. The idea is to write a light field viewer for light field datasets. Useful resources will be the datasets available at the Stanford graphics lab website (look at both the old and new data), as well as in other places. Note that there are some issues with the exact format of the Stanford data, and you may need to do some reverse engineering to get it to work (this was somewhat frustrating for students last year). You may want to start with synthetic datasets (see below).

You can also use a raytracer or your path tracer to render synthetic datasets, which can then be more tightly coupled to your viewer. In the initial phase, this may be the preferred method of choice, since the Stanford lif files may require VQ decompression to be implemented in order to read. It is easy to simply render images from multiple viewpoints one at a time. Of course, eventually you will want to use datasets with acquired light field data, captured as actual images. (note that rendering images from multiple views as a precomputation is very similar to the multiple light images that will need to be rendered for the precomputed radiance transfer project. Indeed, similar compression methods can also be used).

Textual material of interest is the original paper, and the image-based modeling and rendering course notes from SIGGRAPH over the years. You may also look for a reference at the light field viewers and code at the Stanford graphics lab website, though of course you cannot copy directly.

5.1 Basics

In its simplest 2-plane parameterization form, a "ray" in a light field is given by its intersection on two planes. Therefore, to render an image, for each ray, one needs to look up the appropriate light field element, based on intersections of the ray with the two planes. There are of course more effective hardware texture-mapping based approaches, some of which are discussed in the original papers, that you may want to think about (but this is not required).

When you find the intersection of the ray with the two planes, make sure to account for if the ray intersects outside the valid regions or behind the viewer (in these cases, the source light field has no data, and the pixel is simply set to the background color). Once you find the intersection, experiment with finding the closest sample, as well as bilinear and quadrilinear interpolation. Which works best?

As a simple basic test, you may want to render your own light fields at low resolution, and compare with direct hardware rendering of objects, to make sure the basic mechanism is working.

5.2 Input Data and Light Field Viewer

Your basic light field viewer will read a source input file and draw the light field. At this time, we don't specify a particular file format for the inputs, but you may want to be compatible with common datasets such as the Stanford light field archive. The elements of the parameters file are:

- *Resolution:* The first line could have the resolution in u v s t, for example 16 16 256 256.
- *Area of light field slice:* This should tell you the locations of where the uv and st planes are respectively. The line could have the form: uv_x^1 uv_y^1 uv_z^1 uv_w^1 uv_p^1 uv_q^1, where the superscript 1 stands for the first corner of the uv plane, and subscripts stand for x,y,z and possibly w homogeneous coordinate. p and q are the corresponding texture coordinates (in the $[0, 1]$ range). You would have a similar line for each of the 4 corners of the uv plane and then 4 corners of the st plane. For example, a line of the form -2 2 -2 1 0 0 defines the "bottom left" corner with texture coordinates $(0, 0)$ and spatial location $(-2, 2, -2)$. This is similar to the way vertex and texture coordinates are specified in OpenGL
- *Data:* Finally, the data must be given. This could be in a separate file. It could simply be the raw uvst data in RGB triples (the size will be the product of resolutions of u,v,s,t times 3).

Finally, you need to implement the viewer. For this purpose, develop some kind of mouse-controlled view manipulation, such a crystal ball interface. Then, for each pixel, generate a ray as in standard ray-tracing, except you actually intersect with the light field planes, and use that to look up the color in the light field data. Finally, you can draw the image on the screen using standard OpenGL (for example, glDrawPixels()).

5.3 Add-Ons

You should implement at least one add-on. The others will be extra credit. One simple idea is to extend the single-slab light field to multiple slabs, so that you have a closed view of the object. Another idea is to implement VQ or another compression/decompression method (VQ decompression was used in the original light field paper, but some of the light transport theory for precomputed radiance transfer is also relevant). Another simple test is to implement various combinations of nearest, bilinear and quadrilinear interpolation, and compare their relative benefits versus cost. Finally, for the more ambitious, you could use partial geometry like in the lumigraph paper to improve the quality of the light field interpolation and rendering.

5.4 Grading

The tentative grading scheme provides 25 points for a basic working demo (with video), 15 points for good documentation and scene selection, 20 points each for the viewer and suitable interpolation method, and 20 points for the add-on.