Instant Radiosity for Real-Time Global Illumination

Implemented by:
Rohith Chandran and Vivek Reddy

Instructor: Patrick Cozzi

University of Pennsylvania

1. INTRODUCTION

We have implemented a system for real time diffuse global illumination, inspired by the concept of *Instant Radiosity* as described by Alexander Keller in his eponymous 1997 paper. Instant Radiosity is a solution to approximate global illumination by tracing rays from the light source to the scene, letting them bounce around to a specified depth and creating subsidiary light sources at each intersection point. We have implemented such a technique using the Compute Shader, and render the resulting scene using Forward shading. Extensions can be made to render the scene using one of the newer rendering techniques, such as Tiled Forward Shading.

2. RELATED WORK

Global illumination is a family of techniques for approximating real-life lighting situations in computer graphics. In motion pictures, where visual fidelity reigns supreme, techniques such as Monte Carlo path tracing or Photon mapping might be used. However, these techniques consume much time and processing power for rendering, and are therefore not applicable in a real-time setting, like in games, where a framerate of at least 30 frames per second should be maintained to keep up the illusion of motion.

Keller's 1997 paper introduced a method adapted from the Radiosity technique, which he called Instant Radiosity. The idea in this approach (as in Radiosity) is to model light as a source of energy which distributes its energy outward to several surfaces. At any given point, energy from the light can arrive directly from the light or from any of the other surfaces visible to both the light and the given surface. To this extent, point lights (or, *virtual point lights*, as they are called) are created at scene points visible from the light. During final shading, the entire set of actual and virtual lights are iterated over, summing up their contributions and accumulating them to construct the final image.

3. TECHNICAL APPROACH

3.1 Overview

We employ the Compute Shader to perform ray tracing and the creation of virtual point lights in the scene. The Compute Shader became part of the GLSL core specification with version 4.3, and our primary motivation for employing it was to use a solution that would work on graphics cards from all major vendors, as opposed to proprietary solutions like CUDA. OpenCL was not chosen even though it was a viable alternative because the ray tracing that we perform is fairly lightweight, and the Compute Shader was designed for such purposes. Moreover, it is more tightly integrated into the OpenGL workflow than OpenCL (through CL/GL introp) . We based our code on the Deferred Shader we already implemented, and added render paths to it.

After loading the scene, we trace rays from each light source to all of the scene objects, finding

the intersection points, and creating virtual point lights (VPLs) at these points. Then, during the final shading pass, we evaluate each light (actual and virtual), calculating its weighted contribution to the scene, and shade the fragments appropriately.

Every actual source of light will have unit intensity, which we distribute equally to all of the virtual light sources created from it. Thus, we obey the principle of transfer of energy, which is the basis of radiosity. The intensity of each light dictates the share of its contribution to final shading.

Note that there will be some loss of energy from the system, in the form of rays that fly off into the distance, not having intersected any object in the scene.

3.2 Data Structures used

We created structures to store information about each ray that will be traced by the compute shader (Ray) and to store information about the location of VPLs in the scene (LightData). The Ray structure consists of an origin, a direction and an intensity value that will be assigned to each VPL that will be created at the intersection of this ray with a scene object. The LightData consists of two elements, position and intensity, which are self-explanatory. Intensity is a `vec4` value in which we store the colour of the VPL in the r, g and b components, and the actual intensity in the a component.

In order to trace rays into the scene using the compute shader, we need a method for transferring data from and to the same. Earlier versions of OpenGL lacked a structure or feature that would map well to this scenario. In version 4.3, the Shader Storage Buffer (SSB) was introduced alongside the Compute Shader. It is a seamless structure for bidirectional transfer of data between the rendering program (the host-side code) and a shader (any shader, not just compute). The Shader Storage Buffer allows for simple and straightforward transfer of large data (such as arrays of structures) to/from shaders.

Our implementation uses the SSB comprehensively. We use it to create the first set of rays for every light which is traced through the scene by the compute shader. Once the compute shader finds the intersection points, it stores these positions in another SSB, which we read from in our fragment shader. The results that the compute shader writes to the SSB are readily available to all the other shaders and there is no need for the host-side code to do any copying of any sort.

3.3 Workflow

A pictorial representation of the workflow of the program as given in the overview is shown in Figure 1. The red shaded stage is the fragment shader stage, blue-shaded is the vertex shader stage and the green shaded one is the compute shader stage.

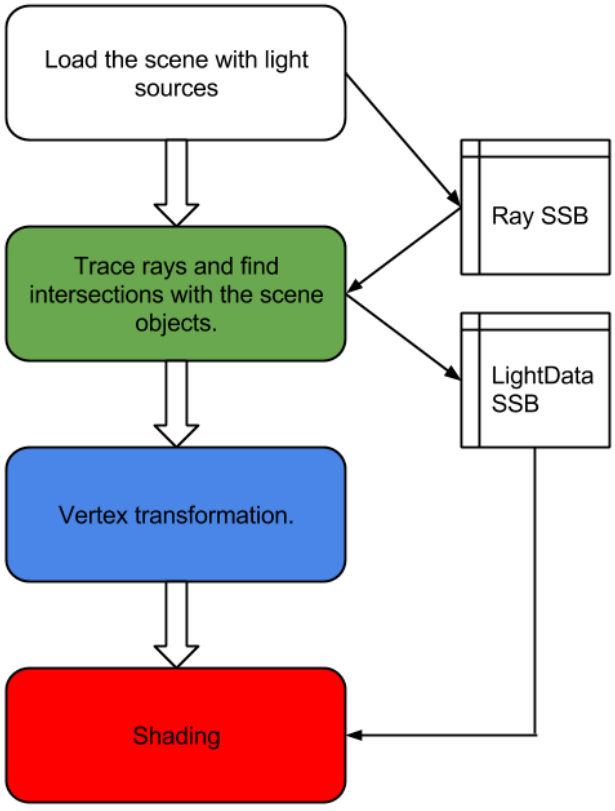

Figure 1: Workflow of our project.

In the workflow, we've shown the compute shader integrated into the pipeline. It must be noted that in reality, the compute shader is not part of the graphics rendering pipeline. It stands alone as a shader program all by itself. In our project, we wait for the compute shader to finish

execution so that the VPL locations may be fed into the fragment shader, where it's used.

4. IMPLEMENTATION AND RESULTS

4.1 Difficulties faced

In the course of implementation, we learnt some hard facts about memory alignment in SSBs, which took away a considerable amount of our time. Essentially, we learnt that graphics cards (or their drivers) **strongly prefer** 16-byte aligned data members in SSBs. Padding is automatically inserted by drivers if any of the components of an SSB is not 16-byte aligned. The structure of the padding is not a part of the official specification, and offset calculation varies between vendors. This can lead to undefined and unexpected behaviour which can vary among cards, vendors and even driver versions.

We had to refactor our code to use `vec4`s to avoid issues we faced with otherwise perfectly working code. Additionally, we found that even for function calls within the shader, it is highly advised to use `vec4`s or built-in types instead of custom structures.

4.2 Results

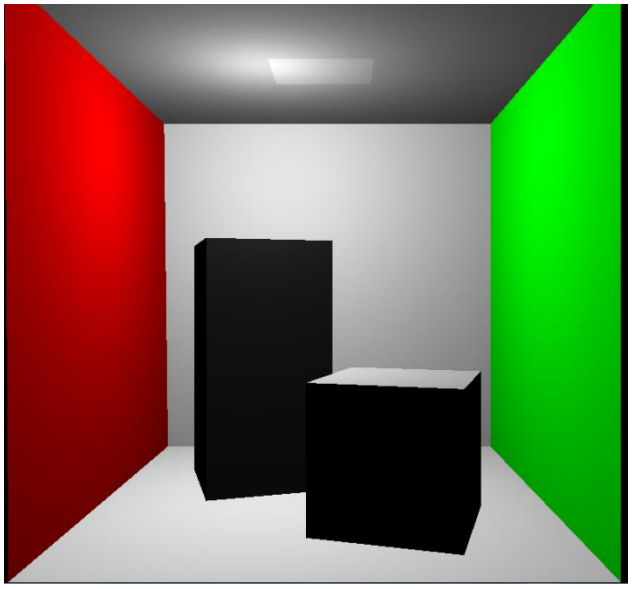

Figure 2: Cornell Box with just direct lighting

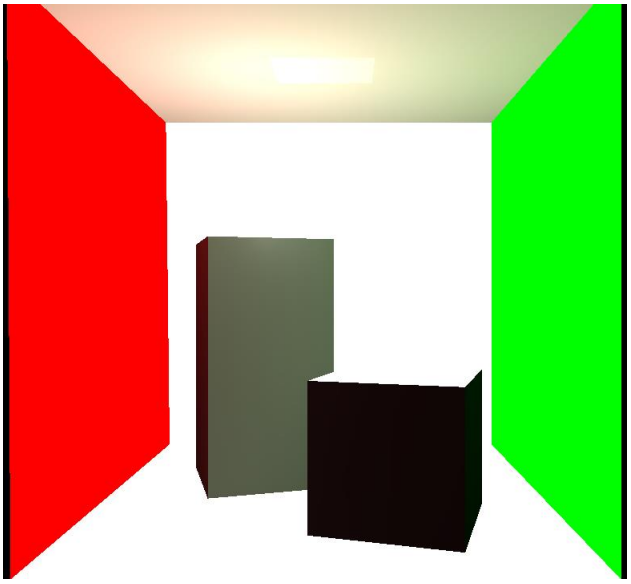

Figure 3: Cornell Box with both direct and indirect lighting, rendered using Instant Radiosity. No tone mapping was used, and the resulting image looks a little blown out.

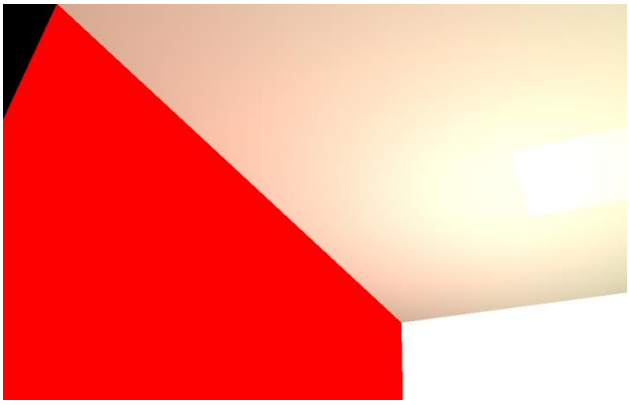

Figure 4: Colour bleeding illustrating Global Illumination

We measured performance on a desktop system having an Intel Xeon processor, 8 GB of RAM and a GeForce GTX Titan, comparing the framerates for both the forward and deferred rendering passes. The results we obtained are as given below:

No of VPL's	Forward Shading	Deferred Shading
128	252.75	280.88
192	176.94	216.57
256	135.73	186.81
320	108.24	160.84
384	92.91	141.29
448	80.76	131.34
512	68.73	120.28

Figure 5: Performance numbers for the scene on a GeForce GTX Titan, using Forward and Deferred rendering paths.

4.3 Improvements/Extensions

The original scope of the project involved implementing alternate rendering techniques, such as a tiled forward or a tiled deferred renderer. Specifically, we aimed to implement Forward+ shading, a specific tiled forward rendering technique as implemented by AMD in their Leo tech demo for the Radeon HD 7000 series.

However, we ran into difficulties and general quirky behaviour with the SSBs and Compute Shaders. One specific example was discussed in Section 4.1. Due to another as of yet unresolved issue, shadow mapping doesn't work in the forward rendering path, while it works perfectly fine in deferred rendering, although both paths use the exact same code and texture to achieve the effect.

Such unexpected behaviour cost us a lot of time in debugging. As a result, although we have code contained in the submission that implements tiled forward rendering, and although it apparently works (validated by reading the values of the SSB from the host side), the results we obtain from the fragment shader are clearly wrong. So, naturally the first improvement would be to find the root of the problem and solve it so that we'd have both tiled forward rendering and Forward+ working.

Tiled Deferred Rendering would be a natural extension of the project, since we already have an implemented deferred renderer, and an almost functional tiled forward rendering.

4. REFERENCES

KINKELIN, M. and LIENSBERGER, C.: *Instant Radiosity: An Approach for Real-Time Global Illumination*.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J. and AILA, T.: *Incremental Instant Radiosity for Real-Time Indirect Illumination*, Eurographics Symposium on Rendering, 2007.

HARADA, T., MCKEE, J. and YANG, J. C.: Forward+: A Step Toward Film-Style Shading in Real Time, in *GPU Pro 4: Advanced Rendering Techniques*, CRC Press, 2013.

BILLETER, M., OLSSON, O. and ASSARSSON, U.: Tiled Forward Shading, in *GPU Pro 4: Advanced Rendering Techniques*, CRC Press, 2013.