IMD3002 Term Project - Radiosity Rendering In Maya

Jake Cataford, Alex Winch

March 26th, 2014

Abstract

This document outlines a plan to implement a radiosity renderer into Autodesk Maya. The implementation includes writing a c++ plugin for Maya using their SDK, and wiring up a UI in Python to configure the renderer. The renderer will be able to render a single frame to a file using Radiosity to simulate global illumination for the scene. This document outlines the process of building such a system, the objectives that should be accomplished as well as a background surrounding the technology.

Contents

Chapter 1

Introduction

This section of the document is to provide more information on the background of the technology that we used in this implementation, As well as provide an overview as to our teams particular approach and development environment, including any overarching dependencies or major considerations that affect our team.

The renderer outlined in this document will be using a approach to simulate global illumination called radiosity. Radiosity is a way to simulate light bounces around the environment by interpreting the amount of albedo any given patch (arbitrarily sized or other) can see in the scene. This method is then recursed for [n] passes to increase the accuracy of the final product.

The renderer itself will be split into two different components: a python script for allowing configuration in the Maya UI, and a C++ renderer dll/bundle plugin to perform the actual rendering to a file. C++ is used for the rendering process because it fits the Maya design patterns more idiomatically, performs much faster, and is far more optimizable, than Python. The plugin will be loaded in a similar way to the way Mental Ray is loaded, and used in a similar fashion.

Chapter 2

Objectives

This section outlines the objectives this project hopes to accomplish. Each objective represents a complete task that is to be completed by the development team in order for the project to succeed. Objectives do not provide implementation details, only generalized statements representing success.

- This project should enable a user to render a scene from within Maya, without using any third party tools or dependencies.
- This project should provide a way for the user to edit and configure the renderer to their personal preference.
- This project should allow approximation of global ilumination through the radiosity technique.
- The project source code should remain in a maintainable, modular state.
- The renderer should run efficiently, algorithms within the renderer should have as low of a complexity as possible to reduce render time.
- The project GUI should be implemented in as idiomatic of a way as possible. Following the design patterns that the Maya User Interface adheres to.
- The renderer will render to a [single node] only to avoid spending too much time dealing with threading implementations.

Chapter 3

Method

This section details the objectives in the previous section, and decorates them with implementation details. The details in this section are not final, but represent the plans for implementation based on our current knowledge of the frameworks provided.

3.1 The Renderer

The renderer is a procedural c++ program that harnesses parameters from our GUI and scene information from Maya, then compiles it into an image file by traversing the data through a procedure. This program is not Object oriented because the nature of a renderer is procedural, and there is nothing to gain from having an application state. The renderer processes the data through a variety of algorithms to render the scene geometry and complete the rasterization pipeline. Below, you will find examples of the steps needed to produce the final raster image.

The Technique requires reducing the scene's surfaces into patches, where each patch could be represented by a struct as follows:

struct Patch: Vec3 emission; Vec3 reflectivity; Vec3 incidentLight; Vec3 excidentLight;

Where each property is a color represented by a 3 Dimentional Vector (R,G,B).

To render the scene, consider this process:

- Load the scene.
- Divide each surface in the scene into patches.
- For each point light, raycast randomly around the scene and initialize the patches with some emission value.
- For each area light, give all patches belonging to the light the emission value of the light.

- For each ambient light, add that light value to the emission value of all patches.
- For each patch, set patch emission to be the average albedo and color value of the pixels in the patch.
- For each pass, loop through the patches in the scene and render the scene from the point of view of the patch. Record the sum of the render to the patch. Then calculate excident light via Incident light by reflectance, plus the previous emission value.

Directional lights require a different method of representing their influence on scene light. We take the normal of the patch, compare it to the direction of the directional lights in the scene, then set the render background (non-occluded light) with an emission value of the dot product of the normal and light direction. (Lambertian diffuse model).

Splitting the main surfaces into patches is another process in itself. The patches are not explicity linked to geometry, but each patch does belong to a particular face on the model. During the per pixel lighting stage, we consider the patch emission (and interpolation) for each pixel we render.

3.2 The GUI

Chapter 4

Design Sketches

This section is used to show off some of our GUI design concepts. This content is due to change at any point if we discover limitations within our program.