mjray: Extended Raytracer
CS 488 - Computer Graphics: Final Project
Date: July 20, 2006
Name: Mike Jutan
Userid: mjljutan
ID: 20079294
Section: 002
Professor: Gladimir Baranoski

Contents

1 Objectives

Due: Thursday, July 20, 2006.

Name: _____

User ID: _____

Student ID: _____

A4 Extra Objective: Reflection

___ 1: **Uniform Spatial Subdivisions (Grid Creation, WCS Axis-Aligned Bbox Transformations and Voxel Traversal):** The Raytracer will use Uniform Spatial Subdivisions as a method to increase efficiency. This requires 3D grid creation, bounding box code for for all implemented primitives and Voxel Traversal using a 3D version of DDA.

___ 2: **Multi-processing and Analysis:** The Raytracer will run a number of processes, greater or equal to 1, which maximizes computational speed. This number will be determined by a set of Raytracing experiments, and will be documented in the analysis.

___ 3: **Adaptive Anti-aliasing:** Anti-aliasing is carried out on the scene to remove jaggies.

___ 4: **Refraction:** Snell's law is used to compute the angle of transmission and secondary rays are cast on intersection with translucent objects to produce refraction effects.

___ 5: **Glossy Reflections:** Objects may have a glossy reflection material applied.

___ 6: **Solid Texturing and Procedural Texturing using Noise:** Textures can be applied to primitives, including checkerboard and procedurals using Perlin noise.

___ 7: **Texture Mapping:** Texture mapping has been implemented.

___ 8: **Phong Model using Vertex Normal Interpolation on Triangles:** Vertex Normal Interpolation is used to create smoother surfaces.

___ 9: **Area Light Support and Soft Shadows:** Planar light sources are implemented and extra shadow rays are used to produce soft shadow effects.

___ 10: **Final Scene:** A final, unique scene is created with techniques from Fine Arts. This scene will demonstrate the features of my Raytracer.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

2 Project Goals and Intentions

2.1 Purpose

My passion in Computer Science has been centered on the fusion of Fine Arts and Computer Science for just about as long as I can remember.

As a Computer Science student at Waterloo, I have enrolled in many Fine Arts and Film classes to better my understanding of the Fine Arts world and to further develop my artistic skills. At the time of writing this proposal, I am enrolled in an upper-year Fine Arts independent study course, where I am being taught in a private one-on-one fashion. In this course, I am learning concepts of colour theory, scene composition, reflection and refraction and I am also getting a chance to do lots of modeling, texture mapping and other techniques in Maya.

This explanation I hope provides the reader with a context for my project, and why I have chosen to extend my Raytracer with some Fine Arts techniques and ideas in mind.

I have taken some of the Fine Arts skills that I am developing, and applied them to the Final Scene for my Extended Raytracer Project. The technical requirements for my Raytracer are based quite closely on my artistic goals for this Project, and thus the technical requirements were a direct consequence of solidifying my artistic goals.

Pixar's creative genius John Lasseter is someone who I really look up to, and as he said in his famous quote: *"The art challenges technology and the technology inspires the art."*

I hope that I have been successful in taking John Lasseter's words to heart on this Project by creating some art that doesn't just look like it's been obviously computer-generated. Also, I have been able to successfully figure out some pretty tough technical details in the extremely limited 3 week time frame that we had for this project.

2.2 Topics and Goals

- Raytracer speed enhancements to allow more complex scenes to be rendered.
- Effective glass simulation, using Reflection and Refraction.
- Using Fine Arts techniques to create an artistic Final Scene with good composition and object placement.
- Several extra-features, enhancements and "going the extra mile" on this Project.

2.3 Milestones

The following order was taken to achieve my objectives.
I started with the major speed optimization first, to speed up render times for subsequent objectives.

- Refraction
- Uniform Spatial Subdivisions (Grid Creation, WCS Axis-Aligned Bbox Transformations and Voxel Traversal)
- Phong Model using Vertex Normal Interpolation on Triangles
- Solid Texturing and Procedural Texturing using Noise
- Glossy Reflections
- Texture Mapping
- Area Light Support and Soft Shadows
- Multi-processing and Analysis
- Adaptive Anti-aliasing
- Final Scene

2.4 Personal Organization and Coding Method

For an individual coding project of this magnitude with over 7000 lines of code, organization is incredibly important. I used the following tools to make the sheer magnitude of this project much easier to handle, by solving it in bite-size chunks.

- gedit: For coding
- CVS: For code versioning
- gdb: For debugging
- Ubuntu Linux, release Breezy Badger: For working at home

2.5 Statement

I modeled some beer glasses (pint glasses) and other objects in Maya, a 3D modeling package. I then exported these items as `Wavefront .OBJ` files, and imported them into my Raytracer. This required some extensions to the supplied `OBJ` importer, to allow for vertex normals and UV texture coordinates.

For my Final Scene, I have multiple high-poly count mesh objects on a wooden table, with a picture frame in the background, and I will also perhaps have other common household items on the table. My intent is that I would like to show what appears to be a random assortment of items on a table, something that is so commonplace in everyday living, that many people would simply ignore these items completely. My focus in my Fine Arts courses it to suggest that normal, everyday items can have a lot of beauty to them, if the small details are observed. For instance, light reflection and refraction in glasses is very beautiful, and is quite often completely ignored due to it's mundane, everyday nature. I hope to convince the observer that these types of light interactions with "everyday" objects have a much deeper level of beauty than most people would initially expect.

To implement a Raytracer capable of such goals, I needed to research and implement several specific features. The obvious one was of course a decent glass simulation, which I achieved through reflection and refraction, and by using the Fresnel method to achieve a more realistic combined reflection and refraction effect. To setup the indoor table scene, I needed to implement several interesting solid (procedural) textures, such as wood grain. I used several different functions, such as Perlin noise, to get a decent texture simulation. This kind of indoor scene suggests the need for a hanging picture frame, and thus I also required texture mapping. In the hopes of further increasing the believability and effect of the Final image, I also felt that soft shadows, phong shading and several other graphics techniques were necessary to achieve my artistic vision.

I believe that this is a very interesting concept and I hope that I have convinced the reader that my ideas are worthwhile. In terms of the complexity of this Project, I believe that many of the goals that I made are computationally expensive, and this implied the need for some speed-up Algorithms, which I implemented in the form of spatial subdivisions, by running my Raytracer as a multi-process computer program and a spatial subdivisions optimization which separated the mesh objects into their polygonal components and associated these sub-mesh components with voxels.

I learned many new Graphics techniques while implementing this Project, and I feel that I really understand a lot more about the Physics of light interaction and Raytracing. More importantly, I had a fantastic time creating this Raytracer!! While the overall learning experience is perhaps the ultimate benefit of this project, I certainly appreciated the journey I took to achieve this knowledge. It was a ton of effort, but it's been very motivating to see the graphical results as I have been developing this project. I had such a great time and learned so much while developing *mjray*, that I plan to continue working on my Raytracer after finishing CS 488.

3 Program Information

3.1 Running the program

`./mjray <inputFile.lua>` will Raytrace the scene defined in the file inputFile.lua and output an image as defined in this lua scene description file.

Optional parameters to `mjray` are as follows:

- `-r <[0-n]>`
  Reflection: Maximum reflection recursion depth. This specifies the recursion depth for secondary reflection rays. Defaults to 0.

- `-f <[0-n]>`
  Re**F**raction: Maximum refraction recursion depth. This specifies the recursion depth for secondary refraction rays. Defaults to 0.

- `-g <[0-n]>`
  Grid size: Voxel amount. This value specifies the number of voxels that should be used to subdivide the scene for efficiency purposes, by the uniform spatial subdivisions code. A value of 0 corresponds to *no* voxels, and thus a value of 0 ray traces the scene with the naive "test against every object" Raytracing approach. Defaults to 0.

- `-o`
  Optimal grid size: Calculate and use "optimal" grid size for spatial subdivisions. This flag will override the **-g** flag in the case where **-g** is specified at the same time as **-o**. This uses a method described by [Pharr 2004], to calculate the "optimal" subdivision amount. This is described in the implementation section of this manual. Default is off.

- `-s <[0-n]>`
  Soft shadow sample size. This value specifies the grid dimensions to use for area light sources. A value of 16 samples is specified by the flag **-s 4**, since a grid of 4x4 equals 16 sample rays. Defaults to 1.

- `-a <[0-n]>`
  Adaptive Anti-aliasing sample size. This value specifies the grid dimensions to use for Adaptive Anti-aliasing. A value of 16 samples is specified by the flag **-a 4**, since a grid of 4x4 equals 16 sample rays. A value of 0 will disable Adaptive Anti-aliasing for the scene. Defaults to 0.

3.2 Communication

The basic flow of information is as follows.

Input :
`Wavefront .OBJ` file(s), describing polygonal mesh objects. A scene description file, written in `LUA`, which uses an `OBJ` importer to extract the necessary data from the given `OBJ` file(s).

Interaction :

The Raytracer will not take any user input after the Raytracing process has been started.

Output :

A rendered image of the specified size, meeting the scene specification as supplied by the given `LUA` file. *mjray* will output several important details throughout the Raytracing calculations. The user will be notified via *mjray*'s progress indicator, as to how much of the final scene has been Raytraced. Incremental file writing will be performed.

When the Raytracing is complete, the multiple processes that *mjray* started, will join and combine their pieces of the final image into one file, and temporary image files will be deleted. On completion, *mjray* displays the render time, and the number of rays (primary, secondary, shadow) that were cast in total.

3.3 New LUA Commands

mjray supports the following new LUA commands:

- `gr.arealight(<initial_pos>, <colour>, <falloff>, <corner_point1>, <corner_point2>)`
- `gr.checkerboardtexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.extendedmesh(<name>, <use_uv_coordinates>, read_extended_obj('filename.obj'))`
- `gr.filetexture(<filename.png>)`
- `gr.fresneldiectricmaterial(<kd>, <ks>, <shininess>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`
- `gr.glossyreflectivematerial(<kd>, <ks>, <shininess>, <reflectivity>, <number_of_reflection_samples>)`
- `gr.perlinnoisecosineinterptexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoiselininterptexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoisemarbletexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoisewavytexture(<colour1>, <colour2>, <texture_scale>, <noise_multiplier>)`
- `gr.perlinnoisewoodgraintexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.reflectivematerial(<kd>, <ks>, <shininess>, <reflectivity>)`
- `gr.reflectrefractmaterial(<kd>, <ks>, <shininess>, <reflectivity>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`
- `gr.refractedmaterial(<kd>, <ks>, <shininess>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`

The member function `set_material_texture(<texture>)` was added to the material type in Lua.

4 Code Locations

4.1 Organization of directories

Source `.cpp/.hpp` files and the `Makefile` are located in A5/src/. The executable `./mjray` and `README` are located in A5/ directory. In A5/data/, there are many `LUA` test scripts that were used to test features of my Raytracer. The resulting images are also in A5/data/ and will be used during the presentation of *mjray* to demonstrate it's capabilities. At the time of printing this document, I am not entirely certain if I will have an extra few days after completing the code to create an Animation. I would like to, but this depends on the complexity and amount of time that I would like to spend on my Final Scene. If I finish an Animation by the due date, I will include this in the A5/data/ directory as well.

4.2 Code Map

algebra.hpp, cpp
: Contains the definition of Matrix, Vector, Point and Colour objects. This is very similar to the supplied file from A4, with some extra extensions and useful overloaded operators.

image.hpp, cpp
: Unmodified from the provided code from A4. The Image class allows for the reading and writing of `PNG` image files.

light.hpp, cpp
: Definition and methods for point lights and area lights. Calculates and returns jittered points for area light calculations.

lua488.hpp
: `LUA` includes, as given in A4.

main.cpp
: Uses C++ `getopt` to parse the command-line options and check for errors in command-line input. Creates the Raytracer object and sends user-specified command-line options to the Raytracer class.

material.hpp, cpp
: Uses Polymorphism with virtual methods to define all types of materials available in *mjray*. Contains code to determine the colour of a point on an object given diffuse and specular properties. Also may optionally contain a Texture object pointer for UV Mapping and Solid Texturing.

mesh.hpp, cpp
: Contains the Polygon and Mesh classes, both of which extend the Primitive class. Contains code for calculating vertex normal interpolation and interpolated UV texture coordinates for extended meshes. Bounding Boxes for Polygons and Meshes are generated here as well as the Intersection code for Polygons and Meshes.

perlin.hpp, cpp
: Implementation of 3D Perlin noise, based heavily on Ken Perlin's reference implementation. [Perlin, 2002].

polyroots.hpp, cpp

Robust Quadric, cubic and quartic root solvers, as given for A4. I do not use this in my implementation.

primitive.hpp, cpp

Contains the implementation of Bounding Boxes, which are used in several places throughout the Raytracer. They are used for the axis-aligned bounding box conversion code used by spatial subdivisions, they are used simply as primitive bounding boxes for intersection tests, they are used to associate primitives with the Voxels that contain them, and they are used for the calculation of the scene extents. Primitive also includes ray-primitive intersection code, surface normal generation and UV coordinate generation for sphere, cube, nonheirSphere and nonheirCube primitives.

raytracer.hpp, cpp

This file contains the "meat and potatoes" of the Raytracer. The main ray tracing algorithm is contained within this file, which casts primary rays, checks for shadow intersections, and recursively casts reflection, refraction and glossy reflection rays when necessary. This file contains the Voxel grid generation methods and the 3D-DDA style algorithm which traverses the Voxels. This file contains the code to create multiple processes, and to join them on their completion and merge the resulting image files together. This file also contains the Adaptive Anti-aliasing code.

scene.hpp, cpp

Data structure class which stores nodes in a DAG. This takes care of "pushing" and "popping" matrices and copying Geometry nodes for the initial preprocessing required for hierarchical raytracing. This file also contains the intersection code matrix translations required for the naive (non-Voxel) method of raytracing.

scene_lua.hpp, cpp

This is the interface between C++ and `LUA`. All new `LUA` commands are defined here.

texture.hpp, cpp

Uses Polymorphism with virtual methods to define all types of textures available in *mjray*. Contains a virtual function to determine the texture contribution at a given intersection point on a surface. This function is called from the Material class if the material has a Texture defined on it. This class handles both UV Textures and Solid Textures.

voxel.hpp, cpp

This class defines the Voxel object and contains a list of Primitive pointers that are associated with the Voxel. The Raytracer class contains an array of Voxels which make up the grid subdivision scheme.

5 Implementation of Objectives

With help from Professor Baranoski, I was able to come up with a list of objectives that were well-balanced in terms of complexity, and still maintained a high amount of personal interest to me. I was able to achieve these 10 objectives in the given time frame, giving me the ability to maximize the effectiveness of my final scene.

I will now outline the technical aspects and implementation details of my objectives and discuss some of the practical issues and concerns that I faced when implementing these objectives. I will also include references here to technical papers that I used in the implementation of these objectives.

5.1 Uniform Spatial Subdivisions

Relevant code is located in `raytracer.cpp` and `raytracer.hpp`.

This was my hardest objective, and took several full days to complete. I therefore was quite careful about how I organized my coding style with this objective. I made sure to "unit test" each subsection of this objective with great scrutiny, so as to ensure that portions of this objective were working correctly and that I could rely on this fact when writing subsequent portions of this objective.

I chose to implement this objective first, so that the speed enhancement given by this objective would greatly benefit my development of future objectives. This turned out quite well, as my Raytracer was quite fast when I often needed to render multiple test scenes to test other objectives.

5.1.1 Grid Creation

Initially, I started with the Grid Creation code. This is contained within the Raytracer class in `Raytracer.cpp`.

Raytracer contains an array of Voxel pointers in a 1-dimensional array which is defined to have a size of `[grid_size*grid_size*grid_size]`. Voxels are accessed by an offset into this 1-dimensional array using this form of index for `voxel(x,y,z) = grid[x + y*grid_size + z*grid_size*grid_size]`. Note that grid size is a command line parameter that can be defined by the user with the `-g` flag. Also, grid size can be specified to be the "optimal" grid size if the `-o` flag is used.

As mentioned in *Physically Based Rendering, pp. 185-186*, the grid resolution is often chosen to be a scalar multiple of the cube root of the number of primitives in the scene [Pharr 2004]. In *Physically Based Rendering*, Pharr explains that this scalar multiple is chosen empirically, and that for his implementation, he used a value of 3. Through some empirical testing, I determined that a value of 3 worked quite well for my implementation as well, so this is the value that I am using for the optimal grid size calculation.

So to re-iterate, if the user chooses the `-o` flag, it overrules the `-g` flag if this is also specified, and the grid size is chosen to be the estimated optimal grid size. This is chosen using the following formula: $3 * \sqrt[3]{N}$, where N is the number of polygonal faces present in the imported Mesh objects.

5.1.2 WCS Axis-Aligned Bbox Transformations

Each object which extends the Primitive class must contain a bounding box. On construction, each Primitive calculates their own bounding box extents and stores their bbox as a member variable. This comes in handy in 2 specific circumstances: when I need to calculate the scene extents, and when I need to associate primitives with the voxels in which they are contained.

Firstly I will describe the scene extent calculation. Raytracer calculates the scene extents (the width, height and depth of all the primitive objects in the scene), by querying each GeometryNode member of the flattened scene DAG. This is carried out by iterating over the allObjects list (which is a `std::list` of GeometryNode*'s.) Raytracer asks each GeometryNode* for it's WCS AABB, and then does a simple min/max test over all of the returned WCS AABB's.

Aside: Note that in the following sentences I will use the terms "MCS AABB", "WCS OBB" and "WCS AABB". These terms stand for the following: "Modeling Coordinate System Axis-Aligned Bounding Box", "World Coordinate System Oriented Bounding Box", and "World Coordinate System Axis-Aligned Bounding Box".

Each GeometryNode* is queried for it's WCS AABB, so it must change it's stored bounding box (which is defined in MCS) to a WCS aabb. This is taken care of by the `getWCSAxisAlignedBoundingBox` method in the GeometryNode class. Each of the 8 bounding box corners are multiplied by the required transformation matrix, which gives an Oriented Bounding Box in WCS. This OBB is then put through the same min/max bbox calculation formula again to transform the WCS OBB to a WCS AABB. This is then returned to Raytracer to help in the creation of the scene extents.

Once the Voxels are created, each Primitive in the Raytracer object's primitive list must be associated (i.e. added) to a list of Primitive pointers in each Voxel that it overlaps. [Pharr 2004] explains in *Physically Based Rendering, pp. 186-187*, that this is easily done now that we already have the world AABB's of each primitive. Instead of transforming Voxels into MCS to intersect with Primitives in MCS, I do these primitive-voxel boundary calculations in World Space instead. This turned out to be a nice way to do this, and it is reasonably easy code to understand. After these calculations are complete, each Voxel contains a list of Primitive*'s which are contained, or partially contained, within the boundaries of it's [the Voxel's] own bounding box.

5.1.3 Voxel Traversal

Once the Voxel grid was created and Primitive*'s were associated with the correct voxels, it was time to write the Voxel Traversal code. I wanted to ensure backwards compatibility with the naive raytracing approach so that I could do some major testing back and

forth between the Voxel traversal method and the naive approach. I therefore decided to separate the Voxel traversal code from the castRays method in Raytracer.cpp where the naive approach occurs, and add another method specifically for this case, this is the method `Raytracer::findNearestPrimitiveInVoxel`.

First I check to see if the incoming ray's origin is already inside the Scene Extent's Bounding Box or not. If it's not, I intersect the ray with the scene extent Bounding Box to find this t-value, and I move the origin of the ray to this point.

I then calculate the initial voxel that the ray is in, and define some necessary setup variables. These include stepX/Y/Z, which are set to 1 or -1 based on the whether we increment or decrement X/Y/Z when we cross voxel boundaries. This is determined by the direction of the ray. I then follow the Algorithm as described in the paper *"A Fast Voxel Traversal Algorithm for Ray Tracing"* [Amanatides, Woo 1987].

I determine the t-value at which the ray crosses the first x, y and z boundaries of the Voxel. The minimum of these 3 values indicate how far along the current ray that we can travel while still remaining within the boundaries of the current Voxel. A delta value for each direction is also calculated so that we know how far along a ray we need to travel to equal the width, height or depth (respectively) of the voxel.

After calculating these values, I loop until I find a voxel with a non-empty Primitive* list. I then intersect the ray with all of the primitives contained within this primitive list, and I return the closest intersection that is still contained within the current Voxel. Note that [Amanatides, Woo 1987] discuss an optimization to determine whether or not the intersection point is within the current Voxel. Amanatides and Woo reduce the amount of comparisons from 6 to 1. In practice, the implementation of this comparison optimization was very messy and did not fit well with the structure of my Raytracer. It required either a double for-loop break, which could be implemented with flag variables or a goto statement, or otherwise it required a call to another function with an argument signature of over 16 variables! This of course in practice was much messier than necessary, and after coding this I decided to simplify my code and I changed back to the 6 comparison method. This method in practice is decently fast, and did not require the code complexity that I required for the Amanatides and Woo method.

This code change allows my rays to take a path through the Spatial Subdivision Boxes, stopping when a valid intersection occurs. This effectively culls a large quantity of ray-object intersections, which is often the majority of the computational expense in raytracing. By this reason, Spatial Subdivisions greatly increased the performance and efficiency of my Raytracer.

5.1.4 Extension to sub-mesh primitives for super-efficiency

The idea of determining the scene extents and then "gridifying" the scene is a great idea - but under the hood it didn't initially work as well as it could, due to some details of my underlying design left over from Assignment 4. The problem was that the basic form of object was a GeometryNode, and this stored a pointer to a Primitive. These Primitives could be

spheres, cubes, or mesh objects. But what about individual Polygons? These were of course contained within their parent Mesh object. This posed a problem for the effectiveness of my spatial subdivisions.

A good example is the rendering of a 2000-poly Venus De Milo mesh object. My testing with this object led to a large re-design of the underlying code so that Primitives (as opposed to GeometryNodes) could be stored in Voxels, thereby breaking a large mesh object into it's individual polygonal components.

This was quite a major change and not part of my initial plan, so I will continue this discussion later on in the Extra Objectives and Enhancements section.

5.2 Adaptive Anti-aliasing

Relevant code is located in `raytracer.cpp`.

Rather than implementing straightforward Supersampling, I implemented an Adaptive method which operates in the following manner:
First, the image is fully Raytraced with only 1 ray per pixel. Following this, and after the multi processes return and combine their results into the final, 1-ray-per-pixel image, I then iterate over these pixels and locate "bad pixels." Bad pixels are defined to be pixels that differ from their surrounding (neighbouring) pixels by more than a certain threshold.
After making a set of bad pixels, the Raytracer then Raytraces these pixels and their surrounding areas with multiple rays, and computes the average colour from this technique.

I chose to use the `STL set` class to maintain the list of bad pixels so that I could ensure that once a pixel has been marked as bad and has been supersampled, it is not supersampled again once it has already been done.

This is a multi-pass algorithm - after computing the set of bad pixels, it supersamples those, and then computes a list of bad pixels again. It subtracts the intersection of the new set of bad pixels with an ongoing set of all pixels that have been supersampled, and runs the algorithm again on this new set. This is to ensure that bad pixels are not supersampled more than once. The new set of bad pixels is also added to the ongoing set so that they will not be supersampled again. This will stop after there are no more bad pixels found in the image, or after a certain number of iterations, whichever makes more sense in practice. By using this Adaptive method, I can concentrate the computational power only on the pixels that really "need" Anti-aliasing more than other pixels, and not waste precious computational time by sending multiple rays through pixels that do not require this kind of attention.

5.3 Refraction

Relevant code is located in `raytracer.cpp`.

To implement Refraction, I started with the method specified in class and in the CS 488 course notes. I calculate a transmitted vector through the surface using Snell's Law as discussed in class. As an extension to this, I combined Reflection and Refraction together by

implementing the Fresnel equations to better simulate a dielectric glass material. This is discussed in the Extra Objectives and Enhancements section below.

The following commands were added to LUA to simulate refractive materials.

- `gr.refractedmaterial(<kd>, <ks>, <shininess>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`
- `gr.reflectrefractmaterial(<kd>, <ks>, <shininess>, <reflectivity>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`

The first material (gr.refractedmaterial), takes that standard Kd, Ks and Shininess terms. It also takes the refraction coefficient for the material outside of it, and for itself. These terms are often referred to as η_i and η_t and standard values for these coefficients are 1.0 and 1.33-1.52 respectively. This material by default sets it's refraction intensity term to 1.0, as there is no reflection required for this material.

gr.reflectrefractmaterial is very similar to gr.refractedmaterial, and gr.reflectivematerial which I implemented for A4. The reflectrefractmaterial takes the same input as refractedmaterial, but also takes a reflectivity intensity term, K_r. This multiplicative term is used to blend the relative effects of reflection and refraction together. Since the reflectivity intensity is specified by K_r, naturally the refractive intensity is specified by $1 - K_r$.

Note that the reflectrefractmaterial material calculates both the reflection vector
$\overrightarrow{r} = -\overrightarrow{v} + 2(\overrightarrow{v} \cdot \overrightarrow{n})\overrightarrow{n}$
and also the transmission vector is calculated.

Note that the transmission vector is cast in a direction as described by Snell's law:
$\overrightarrow{t} = \frac{\eta_i}{\eta_r}\overrightarrow{v} - (\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i)\overrightarrow{n}$

where $\cos\theta_r = \sqrt{1 - (\frac{\eta_i}{\eta_r})^2(1 - \cos^2\theta_i)}$ [Hearn and Baker, 600].

5.4 Phong Model using Vertex Normal Interpolation on Triangles

Relevant code is located in `mesh.cpp`, `scene_lua.cpp` and `readobj.lua`.

To create smoother and more effective mesh rendering, I am computing vertex normal interpolation for mesh objects that have vertex normals defined. These mesh objects are read in using a new OBJ parser for "extended mesh objects" in readobj.lua, and this data is fed into scene_lua.cpp where it is parsed and turned into an extended Mesh object with support for vertex normal interpolation and UV texture coordinates. (More on UVs later.)

The normal at the point of intersection is calculated using the standard Barycentric method as discussed in class. The area of the triangle's face is computed in advance and stored as a member variable of the Polygon object itself. At the time of an intersection, the areas of the three contained triangles are computed and the proportions of these areas over the total area of the triangle face is determined. These factors are multiplied by their adjacent vertex

normal, and summed to give the interpolated normal vector for the given intersection point. This normal vector is then used in place of the standard cross-product normal for lighting calculations and this results in a much smoother, less faceted mesh rendering.

I am using the following to determine the area of a triangle in 3-space. Note that if we specify the triangle with 3 vertices, p_1, p_2, p_3, the area can be computed as follows:

$A(\Delta) = \frac{1}{2}|(p_2 - p_1) \times (p_3 - p_1)|$

since, for vectors $\overrightarrow{u} = p_2 - p_1$, $\overrightarrow{v} = p_3 - p_1$, and θ, we can determine the angle between the two using the following:

$|\overrightarrow{u} \times \overrightarrow{v}| = |\overrightarrow{u}||\overrightarrow{v}||\sin\theta|$

5.5 Texture Mapping

Relevant code is located in `mesh.cpp`, `texture.cpp`, `texture.hpp` and `readobj.lua`.

There are two parts to the following description, UV generation for spheres and boxes, and UV interpolation for extended mesh objects which have UVs defined in their `OBJ` files. The purpose of getting the UV texture coordinate values is so that I can use these values to index into a PNG file using the supplied Image class. This allows for the texturing of a 3-dimensional object with a 2-dimensional image file, as the 2-D image is mapped on or wrapped around the 3-D object.

As described above, mesh objects are read in using a new OBJ parser for "extended mesh objects" in readobj.lua, and this data is fed into scene_lua.cpp where it is parsed and turned into an extended Mesh object with support for UV texture coordinates.

Texture objects, specifically FileTexture objects, for use with UV Textures are created in LUA using the following command.

- `gr.filetexture(<filename.png>)`

Following the creation of the File Texture, this texture can be assigned to an existing material using the LUA command `set_material_texture(<texture>)` which is a member function for materials in LUA.

I will now briefly describe the 3 cases here.

5.5.1 UV Interpolation for Triangles

As described above, if an extended mesh is read into the scene and it has UVs defined in its `OBJ` file, these UVs are interpolated in the triangle intersection code at the time of an intersection. This uses the same standard Gouraud-style Barycentric coordinate interpolation as the vertex normal interpolation code.

The only major difference between the UV interpolation and the Vertex Normal interpolation code is that the UV interpolation code must interpolate both u and v separately, and it is interpolating UV coordinate values across the face instead of surface normals. Otherwise, the code for UV interpolation and Vertex Normal interpolation is relatively similar. Due to

these similarities, and thus the possibility of sharing computed values such as the areas of the three contained triangles on an intersection, the code for both UV and Vertex Normal interpolation is done in the same method in mesh.cpp.

5.5.2 UV generation for spheres

When a ray intersects with a primitive such as a sphere, UVs are not defined as they were with mesh objects.

In this case we need to generated the (u,v) coordinates at this intersection point. UV coordinates for a sphere are calculated using the parametric equation of a sphere with $origin = (x_c, y_c, z_c)$ and radius R:

$x = x_c + R\cos\phi\sin\theta$

$y = y_c + R\sin\phi\cos\theta$

$z = z_c + R\cos\theta$

which gives:

$\theta = \arccos(\frac{z-z_c}{R})$

$\phi = \arctan(\frac{y-y_c}{x-x_c})$

and can be converted to UV coordinates with:

$u = \frac{\phi}{2\pi}$

$v = \frac{\pi-\theta}{\pi}$

since $(\theta, \phi)\epsilon[0, \pi] \times [-\pi, \pi]$. Note that we add 2π to ϕ if it is negative to keep the values of the angles positive.

5.5.3 UV generation for cubes

When a ray intersects a cube, I have yet another case. I used the method for ray-box intersections by [Williams 2005] in the paper "An Efficient and Robust Ray-Box Intersection Algorithm."

I therefore had 6 planes defined and the closest intersection of my ray was on one of these 6 faces. The calculation of the (u,v) coordinates is reasonably straightforward once the implementation details are sorted out. The essential premise is that I want to find out how far along the two axis vectors my intersection point is, and get a ratio for this distance. This is computed by taking the x, y or z component of the intersection point (this depends on the plane that is hit), and subtracting this from the horizontal and vertical edges on one corner of the plane. The horizontal and vertical lengths to the intersection point are stored and divided, respectively, by the length of the entire horizontal or vertical vector of the plane.

This gives the necessary [0,1] ratio value and can be used directly for (u,v) mapping.

5.6 Multi-processing and Analysis

Relevant code is located in `raytracer.cpp`, `raytracer.hpp`.

Another speed-up I added to my Raytracer was multi-processing. Unfortunately the machines in the lab are not dual-core, or even multi-processor machines. I did look into the hardware specifications of the glXX machines although, and it does appear than many (or

all) of these machines are Hyper-threaded.

I thought that therefore it would be an interesting idea to try a multi-process approach, and attempt to let the Operating System handle this as a dual (or multi) process, hopefully harnessing some of the power of Hyper-treading on the processor chips.

This was quite an interesting analysis, as I was expecting slight increase (but not a very large increase) in performance on the hyperthreaded machines with 2 processes running. This was the case and even more interesting was that the results from running 4 processes was exactly as I suspected. 2 processes ran slightly faster than 1 process, and 4 processes actually ran slower than just 1 process! This is very much what I was expecting to see for a single-processor hyperthreaded machine, as when 4 processes are fighting each other for CPU time, the result is somewhat like the phrase "too many cooks in the kitchen spoil the broth." In this case, the broth is of course the runtime of my raytracer. ;)

In terms of implementation details, I create 2 processes using the system call `fork()`. The parent process gets half of the image to raytrace and the child process is given the other half of the image. The parent then waits for the child to finish using the `waitpid(pid_t childId, ...)` system call, and thus the parent "joins" with the child process when the child is complete it's half of the image. At this point, the two temporary image files from the parent and the child are combined together to produce the final image. The two half-images are then deleted with the `unlink()` command. This is useful as once the complete image is saved, there is no need to keep the half-images around on the hard disk.

5.7 Solid Texturing and Procedural Texturing using Noise

Relevant code is located in `texture.cpp` and `texture.hpp`.

Like most people my age, I was really excited when the T-1000 melted up and out of the checkerboard-textured floor in the film Terminator 2.

As an homage to this amazing historical Special Effects sequence, I thought it would be necessary to implement the Checkerboard texture. As described by Pharr in *Physically Based Rendering, pp. 542-543*, Solid Checkerboard is a very simple texture. This was in fact quite easy to implement.

Since this alone does not make up much of an objective, I decided to also implement Procedural Texturing, using a form of noise function. I used Perlin's Noise Function, and heavily based my code on his reference implementation at the website `http://mrl.nyu.edu/~perlin/noise`, which is a direct reference implementation of Perlin's paper. This seemed like a good idea since I don't think there are many people in the world who would write code for Perlin Noise better than Ken Perlin himself! The following website was also quite useful in determining some functions for wood texture and/or any other procedural textures that I felt might be necessary for my final image, or were just pretty cool and I wanted to add them to my Raytracer. `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`

The following are the available solid textures in *mjray*:

- `gr.checkerboardtexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoisecosineinterptexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoiselininterptexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoisemarbletexture(<colour1>, <colour2>, <texture_scale>)`
- `gr.perlinnoisewavytexture(<colour1>, <colour2>, <texture_scale>, <noise_multiplier>)`
- `gr.perlinnoisewoodgraintexture(<colour1>, <colour2>, <texture_scale>)`

Each texture takes two colours which are interpolated between using either linear interpolation (for perlinnoiselininterptexture), or cosine interpolation (for all the other solid textures.) Each texture takes a texture scale value, since it is common for the object to be too large in terms of 2-D texture space. The object therefore needs to be "scaled down" in terms of texture space, so that the texture appears as a larger size on the 3-D object. Finally, the perlinnoisewavytexture also takes one more parameter. This is a noise multiplier function, which is added in the calculation of the wavy texture function to produce different results. I liked the different possibilities of texturing available when this value was varied so I decided to parameterize it as well.

The essential nature of Perlin Noise is as follows. The intersection point is given in world coordinates to the material, and this intersection point is then passed onto the texture object. This (x,y,z) value is passed into the Perlin Noise function. The Perlin Noise function takes multiple linear interpolations of the corners of a lattice that contains this (x,y,z) point. The purpose is that smooth transitions are created in the randomized values, with a low memory usage and with an inexpensive computational cost. Perlin noise gives random values with local similarities, so as to achieve a gradual noise effect. This is fantastic for solid texturing, and this is why I chose to use Perlin Noise.

Note that following the call to the Perlin Noise function, another function can be run on the result of the noise function to get different kinds of solid textures. This is how I handle the different kinds of perlin noise materials defined above. For example, perlinnoisewoodgraintexture is defined as follows:

$t(p) = noise(p) * 20.0$
$t = t - floor(t)$

and then cosine interpolation is carried out between the 2 supplied colours to the perlinnoisewoodgraintexture material, and using the value t which is [0,1] as required.

5.8 Area Light Support and Soft Shadows

Relevant code is located in `light.cpp`, `light.hpp`, and `raytracer.cpp`.

The following code allows for area lights in the LUA scene description file.

- `gr.arealight(<initial_pos>, <colour>, <falloff>, <corner_point1>, <corner_point2>)`

These parameters are defined as follows: `<initial_pos>` is the initial position of the light, `<corner_point1>` is a different corner of the light and `<corner_point2>` is the opposite corner of the light. A good way to think about this is that if we want an area light which is parallel to the ground plane, let's say that `<initial_pos>` is set to (x,y,z). To keep the area light parallel to the ground plane, we should vary x in corner1 and z in corner 2, leaving all other parameters as-is. Therefore setting `<corner_point1>` = (x',y,z) and `<corner_point2>` = (x,y,z') gives a good result. Note that `<colour>` and `<falloff>` are defined the same as they are for the point light command in A4.

Given that 3 points are coplanar in 3-space, we know that `<initial_pos>`, `<corner_point1>` and `<corner_point2>` are always co-planar. Using this knowledge we can compute two vectors to define the sides of the area light.

$vec_1 = (cornerOne - initialPos)$ and $vec_2 = (cornerTwo - initialPos)$. Note that with this definition, we can compute any point on the area light source using the equation $p = initialPos + (s * vec_1 + t * vec_2)$, where $s, t \epsilon [0, 1]$.

Note that if just p is returned then we will get lots of banding in the area light shadows. This can be solved by jittering the rays, to replace the banding with noise which is a better solution as the noise can be smoothed out nicely with Anti-aliasing or by increasing the number of shadow rays cast at the area light.

5.9 Glossy Reflections

Relevant code is located in `raytracer.cpp`.

To make more interesting reflections, I added the capability to specify a material as having "Glossy", or "Diffuse" Reflections. This is specified with the following `LUA` command:

- `gr.glossyreflectivematerial(<kd>, <ks>, <shininess>, <reflectivity>, <number_of_reflection_samples>)`

Kd and Ks and shininess and reflectivity are as described earlier for other materials. The interesting parameter here is `<number_of_reflection_samples>`. This allows the user to control how many recursive glossy reflection rays are cast. Note that I only calculate glossy reflection rays on the first reflection recursion because otherwise the computation gets too expensive. Also, the effect of glossy reflections inside other reflections (if I did send glossy reflection rays when at higher recursion depths), is not noticeable enough of an effect to warrant the huge increase in computational time.

The code for glossy reflections is reasonably simple. I get two orthonormal vectors to the reflected ray. I then use the C++ random function `drand48()` to get random scale amounts s and t where $s, t \epsilon [0, 1]$. At this point I ensure that these will still create a perturbed reflection ray that is still within the reflection cone, as defined by the amount of reflectivity for the glossy reflective surface.

I then recursively cast `<number_of_reflection_samples>` perturbed reflection rays into the scene. The result is a soft, blurry reflective surface.

5.10 Final Scene

For my Final Scene, I will be modeling objects in Maya, a 3D Modeling and Animation software package. After modeling and tesselating the objects, I will convert all the models from Maya to `OBJ` format, then import them into my Raytracer using the extended `OBJ` importer.

At the time of printing of this manual, I have modeled a table in Polygons in Maya and a couple of different beer glasses in NURBS. Since the glasses are in NURBS, I have to tesselate the glass in Maya and then make sure to output vertex normal information to the `OBJ` file when I export it from Maya.

Following this, I will apply Fine Arts concepts to arrange the objects and create the scene in a compositionally well-designed manner. I will use the features created in my Raytracer to create an image which I hope to be quite realistic, and which will display the features that I have created.

6 Extra Objectives, Optimizations and other Super Cool stuff

6.1 Fresnel Reflectance

Relevant code is located in `raytracer.cpp`.

I wanted to make really cool glass simulation, so I decided that the Fresnel equations would be necessary for my Raytracer project. I created a new material to simulate Fresnel Dielectric materials, this is available through the `LUA` interface as well.

- `gr.fresneldiectricmaterial(<kd>, <ks>, <shininess>, <`η_i`: incident_refraction_index>, <`η_t`: refracted_material_index>)`

Note these argument values are the same as defined above in the Refraction description. As mentioned in *Physically Based Rendering, pp. 419-420*, Pharr explains that if we make the assumption that light is unpolarized, the Fresnel Reflectance equations are simplified to the average of the squares of the parallel and perpendicular polarization terms.

A close approximation to the Fresnel reflectance formula for Fresnel dielectric materials is:

$$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}$$
$$r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}$$

where $r_{\parallel}$ is the Fresnel reflectance for parallel polarized light and $r_{\perp}$ is the reflectance for perpendicular polarized light. The Fresnel reflectance for unpolarized light is described in detail in the Pharr book.

Important Note! The fresneldiectricmaterial above does not allow the user to specify a reflectance coefficient or a refraction coefficient. This was done intentionally, because the Fresnel equations are used in my Raytracer to calculate the reflection coefficients based on the viewing angle of the Fresnel Dielectric material. The effect is that for intersections on the edges of Fresnel Dielectric material, there is quite a lot of reflection and for intersections that are not on the edges of the Fresnel Dielectric material, there is little to no reflection and therefore often close to 100 percent refraction. This is a much better simulator of the way glass substances act in the real world, and I am extremely please with the resulting images I was able to achieve with the added Fresnel Dielectric material.

6.2 Animation using John Lasseter's principles of Animation

At the time of printing this documentation, I am unsure of how much time I will have remaining after finishing my Final Scene to work on an Animation that I will render with my Raytracer.

I would very much like to do this, but it depends on the time that I have remaining before the code submission deadline. Either way, in case I manage to fit this in, my idea is as follows:

- Animate bouncing ball in Maya, using spline tools for squash and stretch, slow-in, slow-out, etc.
- Get a chance for some creativity and an interesting use of my Raytracer
- Write a MEL script to export each item to a separate OBJ file, per frame of the Animation
- Write a Python or LUA script to import these OBJs into LUA files, and render each frame one at a time with my Raytracer.
- Compile the images together into a movie file, compressed with an AVI format (eg. DivX), using VirtualDub in Windows.
- Make a Soundtrack for the Animation with sound effects and music, with Sonic Foundry ACID in Windows.
- Attach the soundtrack to the movie and voila! My own animation rendered in my very own Raytracer!!

This is a very exciting idea and I have had the intention to do this as a purely subjective goal if I had time for it. If I run out of time, I will definitely get around to doing this when I have a spare moment, because I think it would be really awesome to render an entire animation in my Raytracer!

That said, how could I possibly render over 300 frames in my Raytracer? Well, with a truckload of speed-ups and optimizations of course! This gives a nice segway to my next topic... optimizations that I wrote and extended optimizations that I did for fun.

6.3 Voxel-Polygon Sub-Mesh Optimization

Relevant code is located in `raytracer.cpp`, `mesh.cpp`, `primitive.cpp`, `scene.cpp`.

As described in an earlier section, after initially testing my spatial subdivisions, it was pretty clear that I needed to do a major 2-day overhaul of the Primitive structure and either switch the direction of the Primitive dependency on GeometryNode, or at least somehow get the transformation matrices into the Primitive class.

Most importantly... I needed to associate *triangles* with Voxels, not just the mesh objects that contained them. This was a big job but after a couple of days, it paid off enormously. I am now getting Raytracing times, with some specific scenes, that are over 100 TIMES faster than with my previous spatial subdivisions. This is due to the guess work that needed to be done to choose an appropriate voxel size when the entire scene is one large mesh. This was a problem because the mesh was added to most of the available voxels, and rays were always checked against every triangle in the mesh! Of course this problem made my spatial subdivision code hardly useful at all for high-poly count mesh objects. Since I wanted to model a bunch of high-poly count object in Maya and then import them into my Raytracer, I really required this functionality.

My testing with the Venus De Milo mesh object led to a large re-design of the underlying code so that Primitives (as opposed to GeometryNodes) could be stored in Voxels, thereby breaking a large mesh object into it's individual polygonal components and exponentially reducing render times for high-poly count mesh objects.

This took my Venus De Milo scene from a render time of over 46 minutes to a render time of just over 2 minutes. This is about a 20 *times* improvement to render time.

6.4 Shadow Ray Voxel Optimization

Relevant code is located in `raytracer.cpp`.

Since my Voxel Traversal code did such a nice job of speeding up my render time once I added the ability to iterate over the Polygons directly rather than just the Mesh objects, I thought I'd better continue to put this code to good use. I was still tracing my shadow rays in a "first hit" manner, iterating over the GeometryNode*'s in the flattened DAG until I found an intersection that was closer than the current light. I would then return early. I figured that this was a good optimization already, since I would not intersect with more objects after the initial intersection.

But, oh my, the Voxel code was blazingly faster than the naive method for shadow rays. I modified the code for Area Lights and for Point Lights so that if the user has selected a spatial subdivisions method, the shadow rays also traverse the scene through the Voxels instead of using the naive method. This change alone made a huge difference.

This took my Venus De Milo scene from a render time of over 143 seconds (after the Voxel-Polygon Sub-Mesh Optimization) to a render time of 35 seconds. This is about a 4 *times* improvement to render time since the Voxel-Polygon Sub-Mesh Optimization, and an improvement of 78 *times* as compared with the initial Voxel implementation.

6.5 Optimal Grid Size Calculation

Relevant code is located in `raytracer.cpp`.

While I was in the optimization zone, I figured I should read a bit further into the description of optimal grid size [Pharr 2004].

As discussed earlier in this manual, [Pharr 2004] describes the optimal grid size in his Physically Based Raytracer to be a scalar times the cube root of the number of polys in the scene. Specifically, [Pharr 2004] describes the optimal grid size as $3 * \sqrt[3]{N}$, where N is the number of polygonal faces present in the imported Mesh objects.

[Pharr 2004] says that a good place to "start testing" grid size is $\sqrt[3]{N}$, which is what I had been using for my optimal grid size. After some experimentation, I found that Pharr's suggested scalar multiplication factor of 3 made yet another large effect on my render time. Using the Optimal Grid Size calculation of $3 * \sqrt[3]{N}$ as opposed to $\sqrt[3]{N}$ this took my Venus De Milo scene from a render time of 35 seconds (after the Shadow Ray Voxel Optimization)

to a render time of 24 seconds. This is about a 1.5 *times* improvement to render time since the Shadow Ray Voxel Optimization, and an improvement of 114 *times* as compared with the initial Voxel implementation.

6.6 Makefile Optimized Build Settings

Relevant code is located in `Makefile`.

As one last final crushing blow to my render times (hehe), I modified the supplied Makefile.

I did not expect a drastic improvement, but this did have quite a huge effect. I changed the -g compiler flag (which creates a debug build) to a -O2 flag, which creates an optimized build. I also played around with processor-specific flags, and the omit-frame-pointer flag but this did not appear to have much of an effect as compared with the change from -g to -O2. Using the Optimized Build flags, this took my Venus De Milo scene from a render time of 24 seconds (after the Optimal Grid Size Calculation) to a render time of 10 seconds. This is about a 2.4 *times* improvement to render time since the Shadow Ray Voxel Optimization, and a *whopping* improvement of 275 *times* as compared with the initial Voxel implementation!! Initially, I did not expect render speed improvements of this magnitude, but these improvements and optimizations layered one over the other really had an astronomical effect.

6.7 Discussion of all optimizations combined together

As mentioned above, I got a *whopping* (I think that's a decent word to use here) improvement of 275 *times* in render speed with my Venus De Milo mesh object, as compared with the initial Voxel implementation.

This is a bit inflated, as likely the initial Voxel implementation runtime was actually slower than just running the naive raytracing approach because the initial code did not support sub-mesh components in Voxels. But all in all, these combined optimizations had a massive effect on render times.

- Uniform Spatial Subdivisions
- Voxel-Polygon Sub-Mesh Optimization
- Shadow Ray Voxel Optimization
- Optimal Grid Size Calculation
- Multiprocessing
- Makefile Optimized Build Settings

Note also that I tested more than just the Venus De Milo scene with these optimizations. I also tested a scene with an imported Maya table and a beer glass that I created in Maya, with a teapot behind the glass. This scene initially rendered a 512x512 image in 321 minutes, roughly 5.4 hours. After adding all of these optimizations to my Raytracer, the same machine is now rendering this table scene at 512x512 in 241 seconds, roughly 4 minutes. This is a

speedup of 80 times, which is fantastic. To mention my Computer Graphics hero John Lasseter again, and his famous quote: *"The art challenges technology and the technology inspires the art."* ... now that I have managed to make the Raytracer 80-275 times faster, I suppose that really just means that in the future I should strive to Raytrace images that are 80-275 times better that what I could Raytrace 3 weeks ago. :)

7 Wrap Up

7.1 Possible Improvements and Future Work

I would assume that it's quite obvious to the reader that I absolutely loved creating this Raytracer, and now that I am "finished", I am going to start thinking of a bunch of additions I'd like to add to my Raytracer when I have the time for it. Some of these possibilities are:

- Advanced Lighting Techniques: Photon Mapping, Caustics, Global Illumination, Final Gather
- Further optimizations: Kd-Trees, OctTrees, Adaptive Methods
- More complicated materials: Plastic, Diffuse Materials
- Animation-Specific scene definition: A scene description method which is better suited to rendering Animations
- Lens model: Accurate Depth-of-Field effects
- Maya .ma file importer utility instead of just OBJ importer
- Multi-layered glossy, semi-transparent surfaces

Oh the possibilities. :) I think it's safe to say that I loved doing this, and I certainly hope to get a chance to do some of these "new objectives" when I have some extra time.

7.2 Code and Reference Acknowledgments

I would like to acknowledge the following code that I heavily referenced.
I acknowledge that I heavily referenced the ray-box intersection code in my primitive.cpp file, from sections 1 and 2 of the paper "An Efficient and Robust Ray-Box Intersection Algorithm." [Williams, 2005].

I acknowledge that the code contained in perlin.cpp and perlin.hpp is based closely on the reference implementation of Perlin noise written by Ken Perlin, and available at
`http://mrl.nyu.edu/~perlin/noise`
and is also based closely on a derived reference implementation by Steve Parker (University of Utah) which is available at `http://www.cs.utah.edu/classes/cs6620/`.

I acknowledge that I did not discover the functions used to generate the noise, marble, wavy and wood textures, although I did experiment with the functions to get interesting effects that I ended up using in my Raytracer. These functions were obtained from

http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
and http://www.cs.utah.edu/classes/cs6620/lecture-2006-02-24-6up.pdf.
I acknowledge the use of LUA parsing code, as written by Matthew Christopher Lausch. Used with permission from the author.

I acknowledge that I did not write most of the code in the polyroots.cpp, polyroots.hpp, algebra.hpp, algebra.cpp, image.hpp and image.cpp, though some of the code in algebra.hpp has been modified and extended.

I acknowledge that my function Perlin::floorDoubleToInt makes use of the double to int floor method as given at http://www.cs.utah.edu/classes/cs6620/lecture-2006-02-24-6up.pdf.

7.3 Thank yous

I would sincerely like to thank my two fantastic roommates, Matt Lausch and Matt Philips. These two guys have a wealth of knowledge and experience, and helped me solidify my goals and objective ideas for this Project early on. I very much appreciate their guidance in the early stages of this Project. Perhaps even more importantly, I appreciate their kindness and shared passion for Computer Graphics. This is incredibly motivating and has made a huge difference to my learning in this course.

Thanks to my sister Norma, my Mom and my Dad, and my friends for not taking it personally when I didn't hang out with them, talk to them, call them, or generally ask them how their day was over the past 3 weeks. I have been working on this Raytracer for (at the bare minimum) 10-12 hours per day.

Thank you as well to my bed and my new Disney/Pixar "Cars" sheets for understanding that I haven't had much time to use them lately, and I hope they understand that I will return to regular sleeping hours again once this Project is handed in. ;)

7.4 Bibliography

Arvo, Jim. "Transforming Axis-Aligned Bounding Boxes." In Graphics Gems, Academic Press, 1990.

Amanatides, J., and Woo, A. "A Fast Voxel Traversal Algorithm for Ray Tracing." In Proceedings of Eurographics '87, G. Marechal, Ed. Elsevier North-Holland, New York, 1987, 3-10.

Bikker, J. "Raytracing Topics and Techniques - Part 1 - Introduction." 2004. URL:
http://www.flipcode.com/articles/article_raytrace01.shtml

Cleary, J.G., and Wyvill, G. "Analysis of an algorithm for fast ray tracing using uniform space subdivision." In The Visual Computer, Springer-Verlag, 1988.

Cornell University Program of Computer Graphics. "The Cornell Box.", 1998. URL:
http://www.graphics.cornell.edu/online/box/

Department of Computer Graphics, “CS488/688 Course Notes”, Spring 2006.

Donald and Baker, “Computer Graphics with OpenGL, Third Edition”, Prentice Hall, 2003.

Elias, Hugo. “Perlin Noise”, 2003. URL:
http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Lasseter, J., ”Principles of traditional animation applied to 3D computer animation.” In Computer Graphics, Volume 21, Number 4. ACM, 1987.

Pharr, M., and Humphreys, G. “Physically Based Rendering”, Elsevier/Morgan Kaufmann, 2004.

Perlin, K. “Improving Noise.” In Transactions on Computer Graphics (Proc. of ACM SIGGRAPH ’02), 2002.

Perlin, K. “Improved Noise Reference Implementation”, 2002. URL:
http://mrl.nyu.edu/~perlin/noise

Ray Tracing News. ”Light Makes Right (Glossy Reflections article)”, Volume 12, Number 2. December 21, 1999. URL:
http://jedi.ks.uiuc.edu/~johns/raytracer/rtn/rtnv12n2.html

Snyder, J.M, and Barr, A.H. “Ray tracing complex models containing surface tessellations.” In Computer Graphics, Volume 21, Number 4. ACM, 1987.

University of Utah, School of Computing. ”CS 6620: Advanced Computer Graphics II”, Spring 2006. URL:
http://www.cs.utah.edu/classes/cs6620

Williams, A., Barrus, S., Morley, R.K., and Shirley, P. “An Efficient and Robust Ray-Box Intersection Algorithm.” In Journal of Graphics Tools, Vol. 10, No. 1:55-60, 2005.