Raytracer Manual
Ray Tracing with Realistic LEGO blocks
Name: Daniel Tchorni
Student ID: 20616973
User ID: DTCHORNI

Preface:

Purpose :

The project focused on extending the existent ray tracer from previous assignment. Adding on several features to render a realistic scene, with LEGO's of course. The LEGOs will be accurate depictions of their real-life counterparts, with the top rounded bezels, and cut out interiors. These LEGO models will build complex structures procedurally.

Statement :

The scene will be set within a room, and within the room where there will be a table with LEGO set on top of it. To make the room more realistic, windows will be added as well as simple furniture pieces to fill the room. This table will feature accurately replicated pieces of LEGO in individual pieces and constructed pieces set together that will be procedurally generated. Additionally, there will be a glass of water on the table with LEGO pieces in it to demonstrate some features. Several different views will be used to generate several outputs, but all rendering the same scene.

To accomplish a successful project, I will start by sketching my scene in more detail from several different angles. From there, the ray tracer must be extended to accommodate for the several features. Thus, beginning by learning the theoretical models of how to achieve those features will be integral in smoothly implementing them, which will be the following step. Then testing for a correct results in each feature and adjusting the implementation to solve any undesired results.

In general ray tracing is an interesting practice and implementing the features listed will be challenging in the given time. To achieve a realistic image would be the goal which is in of itself, a daunting task.

In the end, I will have learned and understood my features listed on the next page thoroughly and gained a better appreciation for artificially rendering realistic images.

Technical Outline :

The initial extra feature that will carried on to this ray tracer but not used as an objective was using multi-threading as an optimization.

The first step and new feature would be to implement cylinders and cones as primitives [1].

Then creating furniture models using the primitives that would include a few tables, a couch and a chair.

To generate realistic LEGO pieces with cutouts underneath and cylinders on top though, a fully functional CSG framework is required. In order to accomplish that, new node types have to be introduced in the hierarchical architecture to account for the boolean connectives. This requires modifications to both the source code and the lua scripts. [3]

To create the complex LEGO fixtures, I will use procedural generation through a stochastic model and map it to the general structure of LEGO pieces. This may also require the modification of a lua script to accompany this feature [2].

Stochastic sampling will be integral in increasing the image quality. This requires mapping a pixel within the grid and sending out a number of additional rays within the pixel space. [6]

Installing and Usage:

Installation :

Within the CS488 environment with the built libraries, extract the Raytracer.zip file in the CS488 environment. Enter the extracted directory, and make a new folder `cmake`. Enter that directory and run `cmake ..` then make. When compilation is finished, exit that directory.

Usage :

Once installation and compilation completed successfully, to run the raytracer call `./raytracer toRender.lua` where to `toRender.lua` defines the mathematical objects you wish to render.
There are several options that can be specified before running the ray tracer. Running `./Raytracer -h` will provide a dialog with options and descriptions. An example output is shown below:

```
Ray Tracer: by Daniel Tchorni
Usage:
[*.lua file]
      -d (exclude diffuse portion of lighting calculation
      -s (exclude specular portion of lighting calculation
      -a (exclude ambient light calculation)
      -b (produce the bounding volumes rather than the meshes
      -sh (remove shadows)
      -ph (Completely disable Phong lighting)
      -th (disable multi-threading optimizations)
      -nth [n] (run with n multiple threads)
      -go (disable Gourard shading)
      -ua (bluedefault all boolean nodes to blueunion operations)
      -db (debug mode -> disables multi-threading and anti-aliasing)
      -ss [n] (enter number of points per pixel to sample with)
-h (Print dialogue)
```

Note: Theses flags will be described in further detail in the Implementation section.

Implementation:

Features :

As was implemented in a previous assignment, all features from the previous raytracer rendition carry forward into this implementation. This includes:

- modeling hierarchical or non-hierarchical primitives (spheres, cubes, and meshes)
- Phong lighting model
- Shadow casting
- Lua script handling

New Features :

All features here will be discussing the objectives that were set out in the proposal and completed.

Extra Feature from First Raytracer - Multi-threading :

This is an optimization feature that provides a significant boost in performance. In order for threads to write to a shared instance of an image, a new class was implemented to be shared across all threads called `RenderObject`. This can be found in Raytracer.cpp/.hpp. The number of threads takes a defualt value of:

$$Threads = numberOfCores * 2 * numberOfSamples$$

By taking the number of cores, you maximize your CPU's usage, but many cores can handle multiple threads on their own, where the standard nowadays is usually two, and by adding sampling we account for the extra samples that are casted. Asynchrony was not an issue until attempting to implement refraction, where locks were required so a static member variable that held the depth of refraction was not accessed multiple times . However, refraction was not completed for this rendition of the Raytracer, thus there are no implementations of asychrony within the code base.

Multi-threading can be disabled by passing the `-th` flag before launching the raytracer. You can also specify the number of threads you wish to use with the `-nth (numberOfThreads)` flag as well. **WARNING:** be careful with the number of threads you want to use, I have experimented with a million threads and it froze the computer! I figured that would happen but was still curious to try it.

Extra Feature - Gourard Shading :

This feature was not listed in the proposal, however given the effect Gourard shading has, I thought it was worthwhile. Gourard shading is applied to meshes as well as cubes. In meshes the normals were calculated based off the average normals of all faces that shared a vertex. The face that made contact was then interpolated the normal based off the normals of the face's vertices. Cubes followed the same approach but with an additional vertex to account for with in the interpolation.

A boolean flag can be passed as a last argument when specifying a mesh in Lua: Ex:
`mickey = gr.mesh('the mouse', 'mickey.obj', true/false)`.
The flag -go disables all Gourard shading for the current run.

Objective 1 - New Primitives :

Cones and cylinders were implemented as new primitives, both as hierarchical or non-hierarchical. Few challenges lied in tackling the circular faces of each one, however several bug fixes have remedied this. In lua create a non-hierarchical cone by (x,y,z represents position/origin of the shape):

```
cone = gr.nh_cone('name',x,y,z,radius,minZ,maxZ)
```

In lua create a hierarchical unit-cone by:

```
cone = gr.cone('name')
```

In lua create a non-hierarchical cylinder by (x,y,z represents position/origin of the shape):
`cylinder = gr.nh_cylinder('name',x,y,z,radius,minZ,maxZ)`

In lua create a hierarchical cylinder by :
`cylinder = gr.cylinder('name')`

Objective 2 - Sophisticated Furniture Models :
Furniture models were built using primitives. This includes a couch, chair, and table. All results can be viewed in the final_renders directory.

Objective 3 - Constructive Solid Geometry (CSG) :
Boolean nodes were implemented to carry out unions, intersections, and differences. They are extremely useful in creating interesting shapes that you can not with a basic set of primitives. In order to implement boolean operations, this required collecting ray intersection information for items entering and leaving the object. This was not implemented initially before. This info is required for both intersections and differences. **Note**: meshes may not behave accordingly when using them in CSG as there are many intersection points possible there and are not accounted for when intersecting them.
In lua to create a boolean node:
`csg = gr.boolean('name',type)`
Type can be either {0,1,2}, each representing {union,intersect,difference} respectively. The boolean operations are applied to the children of the boolean node.

Objective 4 - Modelling Using CSG :
CSG was used to model a 1x1 lego brick with accurate dimensions. This can be done in Lua but concerning procedural generation, it was simply added as a new primitive that can be evoked in a lua script.

Thus the command to create a lego brick is:
`brick = gr.lego('name',w,l,flat)`
The argument w represents the width of the brick, l represents the length of the brick, and flat is a boolean argument that defines whether the brick will be a flat tile or a brick.
Unfortunately there is only one possible implementation completed for this rendition thus the only brick you can make is by:
`brick = gr.lego('name',1,1,false)` Future versions will have more options to create multiple bricks or maybe any time of brick as well.

Objective 5 - Procedurally Generated Lego Structures :
A lua command can create a Lego structure with width,length, height. There is also the option of completely filling out the structure or randomly placing bricks in defined places. The cubes are placed logically. Meaning a cube cannot simply float in the air and defy gravity. All bricks must be attached and in a logical possible manner. By having the brick as a primitive, this eases the process of generating bricks by simply taking copies of a single brick to optimize memory usage.
To generate Lego structures:
`gen = gr.proceduralLego('name',w,l,h,`
`{ { brick1Width,brick1Length },..., { brickNWidth,brickNLength},}`
`{ material1, ... , materialN }`, fill)
Options for bricks are specified by their length and width in a list. This is what the generator can choose from in terms of bricks to place. Materials are passed in as a list as well where the bricks are given a random material from this list during generation. The fill argument tells the generator whether or not to completely fill the given dimensions or leave some empty space.
Given the current version, only one brick can be specified but all other facets of the generator are implemented.

Objective 6 - Refraction :

As mentioned previously this was attempted but not completed for this rendition. Thus future versions will allow for this. As well as reflection and fresnel materials could be implemented in the future.

Objective 7 - Scene :

The scene features all working features available in this rendition. The image is called scene.png and is found under the final_renders directory.

Objective 8 - Photon Mapping Casting :

Not implemented in current version.

Objective 9 - Photon Mapping Gathering :

Not implemented in current version.

Objective 10 - Stochastic Sampling :

Random samples are casted within a single pixel. The default sampling rate is 16. The number of samples can specified with the flag `-ss samples` where samples is the sampling rate.

Debugging :

The flag `-db` configures the Raytracer for debugging by disabling multi-threading and stochastic sampling. The general approach used was to choose breakpoints defined in gdb and with the previously mentioned features disabled, variables and stack frames become exceedingly clear.

Bibliography :

[1] Glassner, Andrew S., ed. *An introduction to ray tracing*. Elsevier, 1989.

[2] Fournier, Alain, Don Fussell, and Loren Carpenter. "Computer rendering of stochastic models." *Communications of the ACM 25.6* (1982): 371-384.

[3] Mann, Stephen, *CS 488/688 Fall 2018*: 130-132.

[4] Mann, Stephen, *CS 488/688 Fall 2018*: 163-166.

[5] Purcell, Timothy J., et al. "Photon mapping on programmable graphics hardware." *ACM SIGGRAPH 2005 Courses*. ACM, 2005.

[6] Mann, Stephen, *CS 488/688 Fall 2018*: 139-143.

Objectives:

Full UserID: DTCHORNI **Student ID: 20616973**

___ 1: Objective one.

Implementing new geometric primitives in a cylinders and cones.

___ 2: Objective two.

Create sophisticated furniture models.

___ 3: Objective three.

Implement a fully functional constructive solid geometry framework with union, intersection, and difference

___ 4: Objective four.

Using CSG, model different realistic LEGO pieces as well as a glass for water.

___ 5: Objective five.

Create LEGO structures that are procedurally generated.

___ 6: Objective six.

Implement refraction.

___ 7: Objective seven.

Render the scene outlined above.

___ 8: Objective eight.

Implementing photon mapping casting.

___ 9: Objective nine.

Implementing photon mapping gathering.

___ 10: Objective ten.

Implement stochastic sampling.

___ A4 Bonus: Multi-threading.

Optimized raytracing by multi-threading.