UNIVERSITY OF WATERLOO
Faculty of Mathematics

RAY TRACER DOCUMENTATION

CS 488
Winter 2017
Gladimir V. G. Baranoski

Prepared by
Sarah Fraser
sm2frase
ID 20458408
April 4, 2017

Table of Contents

1.0 Purpose

The purpose of this project was to design and create a functional ray tracer that explores a variety of rendering challenges. These challenges, as described by the 10 objectives detailed below, are what I believe make a reasonable, well-rounded, complex ray tracer.

2.0 Overview

For this project I have created a ray tracer that uses reasonably complex techniques with the goal of creating high, realistic quality renders. I have done this by implementing 10 objectives, listed below. I feel each objective provided a challenge for myself and adds value to the ray tracer.

For each objective, I researched techniques and algorithms explaining how the objective is solved and implemented. I then implemented each objective to my ray tracer. I then tested each objective using a basic modelling scene that I created in order to highlight the given objective. An example of one of my text models is my glossy reflection model. This scene shows 5 spheres, each with a varying level of glossiness. Not only does this scene show that reflection is function, but it also shows how different glossiness levels are handled in my project.

Finally, I applied all of my techniques to my final scene to fully show what my ray tracer is capable of doing.

In addition to my 10 objectives, I added a few small extra features such as Fresnel Equations, an additional primitive, multi-threading, and a website to show off my renders. By adding multi-threading I was able to render scenes faster, which greatly helped with time management. I created the website to easily display my objective renders and explain each of my 10 objectives at a high level. My webpage also displays my final scene.

I chose to do a ray tracer project because I find the topic very interesting and thought provoking. The idea of casting a ray to determine its colour value is something I never would have thought to do and exploring all of the techniques to get a better render intrigues me more and more each time.

The most challenging part for me in this project was the implementation of refraction. Even though the concept was easy enough for me to understand at a high level, trying to debug and create the correct surface normals both inside and outside of a primitive was very difficult. I also found it hard to determine if my final product was correct as I did not have any refractive materials in real life to compare to.

Another challenge I faced was the management of my time and knowing when to "stop". I am the type of person who will start and finish something in one sitting because I tend to have a single focus mind. I find it easy to stay focused on a single topic for a long period of time, but getting into focus takes me a long time and I tend to lose the depth of focus I had earlier when I have to stop and start. Obviously since this project was so large, I had to take breaks. This proved to be a large challenge for me. However, it was beneficial in the sense that I got to work through this personal problem and still ended up with a project I am very proud of.

After completing this project I can confidently say I have learned a vast amount about ray tracing and the techniques used to achieve it. I also believe I have improved my ability to implement algorithms I have read from papers and books. Finally, I can confidently say I have learned how exciting graphics and especially ray tracing, can really be.

3.0 Manual

3.1 README

A ray tracer used to render x-by-y pixel sized PNG images. Features basic ray tracer elements such as shading, hierarchical node objects, and bounding boxes. Includes 8 additional features listed as follows:

1. Primitives
    - Boxes
    - Spheres
    - Planes
    - Cylinders
    - Cones
    - Meshes
2. Texture Mapping
    - Implemented for Planes, Boxes, and Spheres
3. Bump Mapping
    - Implemented for Planes, Boxes, and Spheres
4. Reflection
    - With varying reflection ray depth
5. Refraction
    - Implemented for Spheres only
    - With varying refraction ray depth
    - Combines with reflection using Fresnel Equations
6. Adaptive Anti-Aliasing
    - With varying division depth
7. Soft Shadow

- Using non-zero rectangular light sources
- With varying shadow ray amounts

8. Glossy Reflection
   - With varying glossy reflection ray amounts
9. Animation
10. Final Scene

The Ray Tracer can be found in the directory `/<download path>/cs488/Project/`

To Build:
```
premake4 gmake
make
```

To Build a Clean Build:
```
make clean
make
```

To Run:
```
./Project Assets/<lua filename>
```

To view renders created by this ray tracer, please visit `http://www.toyboxray.wordpress.com`

3.2 Program Communication

3.2.1 Input

The program takes in one parameter - the LUA file used to describe the scene to render. If no input file is given, the default filename “simple.lua” is used.

3.2.2 Interactions

The program does not require any interactions from the user. While the program is rendering the final image, the render progress is outputted in terms of “pixel colour computed” percentage. Each thread outputs its completed percentages in increments of 5%.

3.2.3 Output

Once the program finishes rendering, the final image will be outputted as a PNG file in the same directory that the program was run from. It will be named the given filename as described in the LUA input file (see LUA File Format for more information)

3.3 LUA File Format

To use the ray tracer program, an input LUA file must be given to provide the program information about the scene being rendered. This information includes the objects and their positions in the scene, light sizes and positions, the camera and “lookat” coordinates, as well as runtime variable numbers such as number of threads and number of glossy rays.

The LUA input follows all LUA language specifications.

3.3.1 LUA Objects

- `gr.node(name)`
  - Creates a basic SceneNode node with identifier as specified by name
- `gr.joint(name, {xmin, xinit, xmax}, {ymin, yinit, ymax})`
  - Creates a basic JointNode node with identifier as specified by name
  - Rotation angles xmin, ymin, etc are not relevant to this program
- `gr.nh_box(name, {x,y,z}, r)`
  - Creates a non-hierarchical box GeometryNode node with identifier as specified by name
  - Box is found with bottom back left corner at coordinates {x,y,z} with lengthheightwidth r

- `gr.cube(name)`
  - Creates a unit cube GeometryNode node with identifier as specified by name
  - Cube is axis-aligned at (0,0,0) with length/height/width 1
- `gr.nh_sphere(name, {x,y,z}, r)`
  - Creates a non-hierarchical sphere GeometryNode node with identifier as specified by name
  - Sphere is centred at coordinates {x,y,z} with radius r
- `gr.sphere(name)`
  - Creates a unit sphere GeometryNode node with identifier as specified by name
  - Sphere is centred at (0,0,0) with radius 1
- `gr.mesh(name, <filepath to .obj file>)`
  - Creates a mesh GeometryNode node with identifier as specified by name
  - NOTE: When providing filepaths, either provide an absolute path or a relative path that coincides with the directory location where the program will be run
- `gr.cylinder(name)`
  - Creates a unit cylinder GeometryNode node with identifier as specified by name
  - Cylinder is centered at (0,0,0) with length/height/width 1 and with the z-axis as its central axis
- `gr.cone(name)`
  - Creates a unit cone GeometryNode node with identifier as specified by name
  - Cone point is centered at (0,0,0) with length/height/width 1 (extending backwards) and with the z-axis as its central axis
- `gr.plane(name)`
  - Creates a unit plane GeometryNode node with identifier as specified by name

- Plane is a z-plane centered at (0,0,0) with height/width 1

- `gr.light({x,y,z}, {r,g,b}, {c0, c1, c2}, w, l)`
  - Creates a rectangular light object with its bottom left corner at position {x,y,z}
  - {r,g,b} indicate the light intensity
  - {c0, c1, c2} indicate quadratic attenuation parameters
  - w and l indicate the width and length of the light
    - A light with $w, l = 0$ is considered a point light with zero area

- `gr.material({dr, dg, db}, {sr, sg, sb}, b, i)`
  - Creates a material object
  - {dr, dg, db} indicates the material's diffuse coefficients
  - {sr, sg, sb} indicates the material's specular coefficients
    - Setting all coefficients to 0.0 will result in a non-specular material
  - b indicates the material's brightness factor
  - i indicates the material's index of refraction
    - Setting i to 0.0 results in a non-refractive material

3.3.2 LUA Object Modifiers

- `<parent_node>:add_child(<child_node>)`
  - Adds child_node as a child of parent_node

- `<node>:set_material(<material>)`
  - Assigns the given material to the given node
  - The material of the node can be changed multiple times

- `<node>:translate(x, y, z)`
  - Translates the node by (x,y,z) amount

- `<node>:rotate(axis, angle)`
  - Rotates the node about the given axis by the angle amount

- `<node>:scale(sx, sy, sz)`
  - Scales the node by (sx, sy, sz) amounts
- `<node>:set_texture(<texture filepath>)`
  - Sets the texture of the material of the given node
  - NOTE: This will set the texture for the CURRENT material of the node. ALL nodes with the same material will also receive this texture. If the material of the object is changed after the texture is set, the texture will no longer be applied to said node
  - NOTE: When providing filepaths, either provide an absolute path or a relative path that coincides with the directory location where the program will be run
- `<node>:set_bump(<bumpmap filepath>)`
  - Sets the bumpmap of the material of the given node
  - NOTE: This will set the bumpmap for the CURRENT material of the node. ALL nodes with the same material will also receive this bumpmap. If the material of the object is changed after the bumpmap is set, the bumpmap will no longer be applied to said node
  - NOTE: When providing filepaths, either provide an absolute path or a relative path that coincides with the directory location where the program will be run

3.3.3 LUA Commands

To render the scene described in the LUA file, include the render command (as described below). Every call to render will render one image of the scene.

```
gr.render(node, filename, w, h, eye, view, up, fov, ambient, lights,
threads, refl, refrac, glossy, shadow, AA_level, threshold, highlight,
AA)
```

where

- **node** is the node and all of its children that are to be rendered in the scene

- **filename** name of the PNG image to be rendered
- **w** image width in pixels
- **f** image height in pixels
- **eye** camera eye location
  - {x,y,z} format
- **view** camera “lookat” location
  - {x,y,z} format
- **up** up vector of the camera
  - {x,y,z} format
- **fov** field of view for the camera
- **ambient** global ambient lighting
  - {x,y,z} format
- **lights** a list of light source objects defined by gr.lights(..)
- **threads** the number of threads used to render the image
- **refl** the reflection ray depth
  - If set to 0, no reflection will be included in the render, even if there exists reflective materials
- **refrac** the refraction ray depth
  - If set to 0, no refraction will be included in the render, even if there exists refractive materials
- **glossy** the number of reflective rays to cast per object intersection where $refl > 0$
- **shadow** the number of shadow rays to cast per object intersection
- **AA_level** the max number of subpixels divisions for anti-aliasing
  - If set to 0, the each pixel will only be subdivided into 4 subpixels
- **threshold** the threshold level for adaptive anti-aliasing

- **highlight** used to highlight where adaptive anti-aliasing would be implemented on a render
    - If AA is true then set to 1 to colour the pixels red instead of doing adaptive anti-aliasing on them. Set to 0 otherwise.
- **AA** turns adaptive anti-aliasing on / off
    - If set to 1, AA is turned on
    - If set to 0, AA is turned off

3.4 Project Organization

3.4.1 Source Code

The src for the ray tracer can be found in the Project directory. The source code is separated into 15 modules.

- **A4** Contains all methods used for ray tracing an inputted LUA scene.
    - Thread control
    - Casting rays
    - World-to-Pixel
    - Adaptive Anti-Aliasing
    - Glossy Rays
    - Refractive Rays
    - Reflective Rays
    - Fresnel
    - Ambient Lighting
    - Finding Intersections (Does not include actual intersection calculations)
- **GeometryNode** Class which contains a name, material, and primitive object. Created via the LUA input file. Parent class is SceneNode
- **Helpers** (Also known as the Intersection Class) Class which contains intersection point, intersection normal, material of the intersected object, uv-mapping values, pu&pv-bump mapping values

- **Image** Class which contains image width, height, and data. Includes methods for loading and saving a PNG image.
- **JointNode** Class which contains joint boundary values. Created via the LUA input file. Parent class is SceneNode.
- **Light Class** which contains light position, colour, width, height, and falloff values. Created via the LUA input file.
- **Main** Calls the run_lua method to render the image provided by the input LUA file.
- **Material** Parent class of PhongMaterial.
- **Mesh** Class which contains mesh vertices and faces. Created via the LUA input file. Parent class is Primitive. Includes methods for calculating intersections with a Mesh object as well as creating a mesh bounding box.
- **PhongMaterial** Class which contains material diffuse and specular values, as well as shininess and refractive factors. Also contains texture and bumpmap (both Image objects) references. Includes methods for texture mapping and bump mapping using uv/pu&pv values. Parent class is Material.
- **Polyroots** Methods for finding roots of quadratic, cubic, etc equations.
- **Primitive** Parent class which contains subclasses of different primitives. All subclasses include methods for calculating intersections with said primitive. Subclasses include:
  - Sphere
  - Cone
  - Cylinder
  - Cube
  - Plane
  - Non-hierarchical Box (Contains position coordinates and size)
  - Non-hierarchical Sphere (Contains position coordinates and radius)
- **Scene_lua** Parser used to read in input LUA file, created objects such as nodes and lights, and call the a4_render method.

- **SceneNode** Class which contains node name, node type, a transformation matrix, and inverse transformation matrix, and nodeID. Parent class of GeometryNode and JointNode.

3.4.2 Assets

All data files associated with LUA files, texture PNGs, and bumpmap PNGs, are found in the Assets folder.

4.0 Technical Outline

4.1 Objective 1: Primitives

This ray tracer can render six primitives: spheres, cubes, cylinders, cones, planes, and meshes. As part of A4, I had to implement the spheres, cubes, and meshes intersection calculations and for this project, I implemented the cylinders, cones, and planes (as a bonus). As this documentation is for the project portion of this ray tracer, I will not explain the implementation of the spheres, cubes, or meshes.

4.1.1 Cylinder Implementation

In order for the ray tracer to render an object in an image, it needs to calculate whether a ray cast from each pixel intersects with said object. If there is an intersection, then the object is in the view of the camera and thus needs to be rendered. Thus, to render a cylinder, I created a check intersection method in the Cylinder primitive class.

In this project, the cylinder primitive is a unit cylinder, centered at (0,0,0), and is axis-aligned about the z-axis. Since calculating intersections with the ray is done in the model space, all calculations were done assuming the cylinder is as described above.

A cylinder has two faces and a curved side. Thus, an intersection needs to be checked for all three. Since the cylinder is a unit cylinder about the z-axis, the formula for the plane of the faces will be $z_1 = -0.5$ and $z_2 = 0.5$. The formula for the curved portion of the cylinder is $x^2 + y^2 = 1$ where $0 <= z <= 1$.

To calculate a possible intersection, I first start with the curved surface and make the temporary assumption the cylinder is infinite (meaning I do not have to worry about bounds on z). Then, just as I did with the sphere, I use the formula for my ray ($r = o + td$, where $d = b - o$), the cylinder equation, and the quadratic formula to solve for t.

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

where $A = d_x^2 + d_y^2$, $B = 2 * a_x * d_x + 2 * a_y * d_y$, and $c = a_x^2 + a_y^2 - 1$

After solving for t, I then bound the infinite cylinder using $z1$ and $z2$. Then, depending on the number of solutions for t, I have three possible cases to handle:

1. No solution means there is no intersection with the curved surface
2. One solution means the ray is tangent to the curved surface.
3. Two solutions means the ray intersections the curved surface.

In the event of case 1), I move onto the next step of checking for an intersection with the faces. In the event of case 2), I check the z value of the point given by t. If the point is within the bounds, then it is a valid intersection point if $t > 0$ (meaning the intersection happens in front of the camera). In the event of case 3), there are four possible subcases that can happen:

1. Both z values are smaller than z_1 or both are larger than z_2 which means the ray intersections with the infinite cylinder but before/after the entire bounded portion.
2. One z value is within the z bounds and the other is outside of the z bounds. This means the ray intersects the curve of the bounded cylinder and one of the faces.
3. One z value is smaller than z_1 and one z value is larger than z_2. This means that the the ray intersects the cylinder such that it goes through both of the faces of the bounded cylinder but not the curve.
4. Both z values are within the bounds and thus the ray intersects the bounded cylinder twice.

In the event of case 1) I can stop the check as there is no possible intersection with the bounded cylinder. For all other cases, I need to determine which intersection point is closer to the camera and is thus “seen” first. However in the event of case 2) or 3), before determining which intersection point is closer, I need to calculate the intersection point of the points intersecting the face(s) instead of the current point(s) on the infinite cylinder. To do this, a third (and fourth for case 3)) t value needs to be found using the formula for the face's plane and equation for the ray.

Face 1: $z_1 = -0.5$
Face 2: $z_2 = 0.5$

To determine which face the ray is intersecting, I must compare the z value from the first two t values. Whatever z value is outside of the bounds if the face intersection point, with it intersecting face1 if its z value is smaller than z_1 and face2 if its z value is larger than z_2. In the case of two face intersections, the corresponding face matching can be done the same way.

Thus, $t_3 = \frac{z_i - o_z}{d_z}$ where $i = 1, 2$ depends on the face being intersected.

To calculate t_4, the same actions are taken as with t_3, except the other face is used.

Then, I replace t_3 (and t_4) with the t value of the point which is interesting the face. For example, if t_2 is intersecting face2, then my two new t values are t_1 and t_3 where t_1 is the bounded curve intersection point and t_3 is the face2 intersection point.

As mentioned before, I have two t values and I need to determine which point is closer to the camera. This can be done by comparing the two values. If both are positive, then the smaller of the two t values is the closer intersection point. If one is positive and the other negative, then the positive t value is the close intersection point (a negative t value means the point is behind the camera). If both t values are negative, then there is no valid intersection and the intersection check can stop.

Finally, for all cases of the ray intersecting the curve, I have a valid t value and a corresponding intersection point. If the intersection point is on the face, then the intersection normal is (0, 0, 1) since the faces are bounded z-planes. If the intersection point is on the curve, then the intersection normal is the x and y values of the intersection point and $z = 0$.

Now, in the event that I run into case 1) when checking for an intersection of the curve, I cannot conclude the ray does not intersect the cylinder at all because the ray can be parallel with the z-axis and thus never intersect the curve, but still intersect the two faces. Therefore, I now need to check for a face intersection.

To do this, I will use the same formula I used earlier to find t_3. This will give me two t values corresponding to where the ray intersects the infinite z_1 and z_2 planes.

$t_5 = \frac{z_1 - o_z}{d_z}$ and $t_6 = \frac{z_2 - o_z}{d_z}$

Then, to check if the intersection points are within the bounded cylinder faces, I use the t values and the formula for the cylinder curve.

$$(o_x + t_i * d_x)^2 + (o_y + t_i * d_y)^2$$

If the result is less than or equal to 1 for both t_5 and t_6, then the ray intersects the faces of the cylinder.

Just as before, the smaller of the t values (given it is positive) gives the closer intersection point. Since the intersection happens with a z-plane, the intersection normal is (0,0,1).

[Roth, 1982] [Glassner, 1989] [Drakos, 1997]

4.1.2 Cone Implementation

In this project, the cone primitive is a unit cone of height 1, with its point at (0,0,0) and its face at z-plane $z = -1$, and is axis-aligned about the z-axis. Since calculating intersections with the ray is done in the model space, all calculations were done assuming the cone is as described above.

A cone has one face and a curved side. Thus, just as with the cylinder, an intersection has to be checked for both. Since the cone is a unit cone about the z-axis, the formula for the plane of the face is $z = -1$. The formula for the curved portion of the cone is $x^2 + y^2 - z^2 = 0$ where $-1 <= z <= 0$.

Just as I did with the cylinder primitive, to calculate a possible intersection with a cone, I will start with the curve of the cone and make the cone temporarily infinite. Then, using the formula for the cone, the quadratic formula, and the equation for the ray, I will solve for t to get the possible intersection points.

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

where $A = d_x^2 + d_y^2 - d_z^2$, $B = 2 * a_x * d_x + 2 * a_y * d_y - 2 * a_z * d_z$, and $C = a_x^2 + a_y^2 - a_z^2$

After solving for t, I then bound the infinite cone using $z_1 = 0$ (aka the point) and $z_2 = -1$. Then, depending on the number of solutions for t, I have three possible cases to handle:

1. No solution means there is no intersection with the curved surface
2. One solution means the ray is tangent to the curved surface.
3. Two solutions means the ray intersections the curved surface.

In the event of case one, there is no intersection with the cone and I can stop the intersection check. This is unlike the cylinder where I had to check for a face intersection because there is no possible way for the ray to intersect the face of the cone and not the curved surface of the infinite cone as well.

In the event of case two, there is one intersection. If the value for t is greater than zero and the z value of the intersection point is within the bounds of the finite cone, then the intersection point is valid.

In the event of case three, the ray intersects the infinite cone in two places. Using the values for t, I then calculate the two intersection points using the equation of the ray. Depending on the z values of those two points, I can have one of 3 subcases:

1. Both z values are smaller than z_1 or both are larger than z_2. This means the ray intersects the infinite cone but not the bounded cone in any way.
2. One z value is less than z_1 or greater than z_2 and the other z value is within the bounds.
3. Both z values are within the bounds of z_1 and z_2

In the event of case 1), there is no valid intersection point and the intersection check can stop.

In the event of case 3), both intersection points are valid and on the curve of the cone. The smaller of the two t values will give the closer point to the camera of the two intersection points. If one of the t values is negative, the other is the closer point, and if both are negative, then the points are behind the camera and are no longer valid.

In the event of case 2), the intersection point with the face needs to be determined before finding the close intersection point. Just as I did with the cylinder faces, a new t value needs to be found using the equation of the face's plane. In this case, the face is on the z-plane $z = -1$, thus

$$t3 = \frac{-1 - o_z}{d_z}$$

This new t value replaces the t value of the point outside of the finite cone.

Then, just as before, I then choose the smaller, positive t value as the point of intersection. If neither are positive, then there is no valid intersection point.

Finally, I need to calculate the intersection normal of the intersection point. Just as with the cylinder, if the intersection happens on the face, then the normal is simple (0,0,1). If the intersection point happens on the curve, then the normal is (x,y,1.0) where

$x = \frac{p_x}{mag}$ and $y = \frac{p_y}{mag}$ where P is the intersection point and $mag = P_x^2 + P_y^2$

[Roth, 1982] [Glassner, 1989] [Drakos, 1997]

4.1.3 Plane Implementation

In this project, the plane primitive is a unit z-plane centered at (0,0,0). This means its width and height are both 1, starting at -0.5 and ending at 0.5. As with the cone and cylinder, calculating the intersection with the ray is done in model space, thus all calculations were done assuming the plane is as described above.

A plane is a fairly simple primitive, consisting of a single face bounded by a height and width. Thus, to check for an intersection, I have to simply check for an intersection with the z-plane $z = 0$ and check that the x and y values of the intersection point fall within the bounds of the plane.

To check for a possible intersection with the plane primitive, I first need to check to see if the ray will intersect the infinite z-plane (aka check to make sure they are not parallel) by taking the dot product of the plane's normal and the ray's direction. If the resulting dot product is zero (or very close to it because of floating point error), then the ray is parallel and will never intersect the plane and thus the intersection check is done.

If the dot product is not zero then I can determine at which point the ray will intersect the plane by solving for t where

$$t = \frac{0 - o_z}{d_z}$$

It is important to note here that the above equation for t is only valid when the plane is a unit plane and the normal remains at (0,0,1). Else, the equation for t requires finding the dot product of the normal and the negative of the ray's origin and dividing it by the dot product of the normal and the ray's direction.

Now, the possible point of intersection is found using the calculated t and the equation of the ray.

To determine if the point of intersection is within the bounded plane, I then compare its x and y values to the max/min x and y values of the plane (which are -0.5, 0.5 each, respectively).

If the x and y values are within the bounds and the t value is greater than zero (meaning the intersection happens in front of the camera), then the point of intersection is valid.

Finally, the normal of the intersection point is just the normal of the z-plane, (0,0,1).

[Roth, 1982] [Glassner, 1989]

4.1.4 Implementation Location in Source

All three new primitive classes can be found in `Primitive.cpp/hpp` as well as their intersection check methods.

4.1.5 Render Examples and Demos

A render that demonstrates the three new primitives that I added can be seen by navigating to `/.../Final Renders/1 - Primitives/prim.png`.

To render the image using the ray tracer, the LUA file can be found in the Assets directory under the filename `prims.lua`.

4.1.6 Challenges and Bugs

One of the biggest challenges I faced when implementing these primitives was determining the normals of the points of intersection, especially when they involved intersections of a face and curve. I not only had to calculate the normal based on which type of surface it was interesting, but also keep track of which point was the closer of the two and thus which normal had to be used. I also had a lot of difficulty trying to determine what the normal would be for the surface of the curve. It look a lot of debugging and doing a large number of renders of moving the light around the object to determine what normal was correct.

4.1.7 Future Additions & Unexplored Areas

Currently, I have only implemented intersection checks for unit cylinders, cones, and planes. I think a good addition to this objective would be to extend the intersection checks for non-unit primitives that are not centered at (0,0,0). I have a little bit of the implementation done already, but I did not complete it due to time constraints.

I also think it would be a good idea to include other types of primitives such as a torus and disk. This not only would provide a new challenge for calculating intersections, but allow for the ray tracer to be even more robust in the kind of scenes it can render.

4.2 Objective 2: Animation

Another one of my objectives for this project was animation. There are many ways I could have approached this, but I chose to do animation by rendering a large number of frames and playing them one after another fast enough to appear like a smooth video.

4.2.1 Implementation

To create my animation, I created a LUA scene of a music box with a dancer on top and a crank on the side. In the LUA file, I added a loop that contained a y-axis rotate dancer command, x-axis rotate crank command, and render the image. I then ran the loop a certain number of times, which each loop rotating the dancer and crank by a small amount and then rendering the image. I then took the rendered images and used an online GIF making software to meld them all together.

To decided to make my animation rotate the dancer and crank 360 degrees and then repeat the animation to make it appear as though the dancer and crank were continuously rotating.

In order to get the smooth animation effect I wanted, while keeping the rate of rotate to a certain speed, I played around with the number of frames I created and at which speed/duration they were played in the GIF. I ended up rendering 40 frames for 3200 milliseconds. This results in approximately 12.5 frames / second. Although this is not the standard 30 frames/second of higher quality animations, I found that the number of frames per second was enough to create a smooth animation without me having to render 96 images (which would achieve 30 frames/second). If I had to render 96 images, the render time would have been far too long and most likely would have resulted in me reducing the quality of the render (no anti-aliasing or soft shadows) to compensate.

4.2.2 Render Examples and Demos

The full animation GIF can be found at the path `/.../Final Renders/9 - Animation/final_animation.gif`.

The animation LUA file can be found in the Assets directory, under the filename `animation.lua`. The variable `num_renders` indicates how many final images to render.

4.2.3 Challenges and Bugs

My biggest challenge when creating this animation was the placement and rotation of the crank. I had to ensure that the rotating part of the crank was rotating about the non-rotating part's x-axis instead of its own x-axis while also ensuring that the rotation action did not move the crank away from the rest of crank/box. Thankfully, using joint nodes helped a lot with this problem.

4.2.4 Future Additions & Unexplored Areas

If I were to add/change this animation I would first change how the crank is rotating. Currently, the upwards part of the crank is rotating about the stationary horizontal part of the crank that comes out of the box. In reality, all parts of the crank would be rotating. Not only would this make the animation look more realistic, but it would have also made the rotating step easier to implement since I can rotate the horizontal part about its own x-axis instead of trying to get the vertical part to rotate about the horizontal part's x-axis.

Another future addition that I would add would be to have the dancer rotate about her foot instead of her centre. Realistically, her foot is the part attached to the box and thus should be what she rotates about.

To improve the animation and add extra moving parts, I would perhaps have other parts of the ballerina move like her arms or lifted leg. It might also be interesting to start the animation with the lid opening and end the animation with the lid closing.

4.3 Objective 3: Adaptive Anti-Aliasing

In A4, I chose to make supersampling my additional feature. Although this worked to improve my image renders, I felt like I could not only do better for the project, but smarter as well. Therefore, I chose to do adaptive anti-aliasing (henceforth known as AAA) for my project, where instead of doing the same large amount of supersampling for every pixel, the amount of supersampling of each pixel depends on the "business" or complexity of said pixel and the colour of objects seen from it. The reason why adaptive anti-aliasing creates a better render in a faster amount of time is because it avoids supersampling pixels that have little to no colour variance with the area around them (examples can be things like a consistent background colour or solid coloured objects with no reflections, shadows, or lighting effects).

4.3.1 Implementation

There are a variety of adaptive anti-aliasing techniques one can use in a ray tracer. The one I chose to implement was explained by Don P. Mitchell in his paper "The Anti-Aliasing Problem in Ray Tracing". Mitchell takes a technique already described by Whitted and improves upon it.

The aforementioned technique is described as follows:

1. For each pixel, cast a ray for each corner of the pixel.
2. Then, compare the colour values for the four colours. If the colours differ by more than a certain threshold, then divide the pixel into 4 equally sized subpixels.
3. For each subpixel, cast four rays from each corner and compare the colours returned. As before, divide into four smaller subpixels if the colour difference is above the threshold.
4. For each subpixel, stop dividing if the colour difference is below the threshold or if the number of divisions has reached the maximum number of divisions.
5. Then, the colour of the pixel is the weighted average of all the rays. The weights total to 256, with higher weights going to the subpixels that have been divided less.

[Mitchell, 1990]

I followed this technique fairly closely, but instead of casting a ray from each corner, I chose to cast a ray from the centre of each subpixel instead. In other words, if I have a 1x1 sized pixel, then I cast rays from (0.25, 0.25), (0.25, 0.75), (0.75, 0.25), and (0.75, 0.75) instead of (0,0), (0,1), (1,0), and (1,1). The reason for this is because earlier in A4 when I implemented supersampling (aka this first step), I found that the final image was very muddy and blurry, with the colours bleeding into each other. The reason for this is because by sending a ray from each corner, two adjacent pixels were using the same calculated colour, causing them to be more similar than they actually were. Therefore, to combat this issue, I chose to cast the rays from the middle of each subpixel instead of the corners.

To compare the colour differences of the returned subpixel colours, I created a method that takes in two colours and computes the difference between their R, G, and B values. If any of the differences are over a given threshold, then the colours are too different. I then used this method and compared all four colours with each other. If any of them were flagged by my method as too different, then I further subdivided said pixel as explained above.

To implement the technique, I created a recursive method that takes a subpixel, cast fours rays equally spaced within that pixel, calculates the colour difference, and either returns the average of the four colours multiplied by their given weights or makes four recursive calls for each of the rays that were cast. The initial call to the method starts with the original pixels x and y values and a weight value of 64. For each subsequent recursive call, the modified x and y values are passed along with the current weight divided by four.

A feature that I feel is necessary to include in this objective is the ability to highlight which pixels AAA will supersample more on (aka the pixels whose colour difference is above the threshold). This way, not only do I have an easy way to debug AAA, but I can also demonstrate to others how AAA works with my ray tracer. Implementing this feature was fairly easy. Instead of making a recursive call when the colour differences are over the threshold, I instead coloured the pixel bright red (1.0, 0.0, 0.0). I then added a parameter into the `a4_render` method called by the LUA parser which allows me to turn the highlighting feature on and off.

In my proposal I mentioned that I was going to determine a good threshold and maximum divisional level through trial and error. In order to do this, I added another parameter into the `a4_render` method that lets me set the threshold amount and the level of pixel division. In the end I found that this was a better option than setting a fixed value because certain renders and images require varying values for threshold and depth. In general though I found a good threshold was 0.1 and a good divison depth was 2.

In addition to the highlight, threshold, and division depth, I also added another parameter to `a4_render`. Since AAA does cast more rays than no anti-aliasing I chose to implement a feature that allows me to turn AAA on or off. This allowed me to debug other features without having to wait very long, as well as render images faster that I did not think needed AAA turned on.

4.3.2 Implementation Location in Source

AAA can be found in `A4.cpp`, with a focus on the method `anti_alias_render`.

4.3.3 Render Examples and Demos

For some examples of AAA in action, refer to the `5 - Adaptive Anti-Aliasing` directory in the `Final Renders` directory.

To render an example of adaptive, a LUA file can be found in Assets under the filename `adaptive.lua`.

4.3.4 Challenges and Bugs

One of my biggest challenges when implementing AAA was deciding how I wanted to compare the colours and determine what classified as "going over the threshold". I experimented with a couple of techniques such as using the average of the colours RGB values and testing their differences, by only comparing one value from each colour (such as only comparing R), and only setting colours as "over the threshold" when two or more differences were larger than the threshold. In the end, most of these techniques weren't sensitive enough or did not pick up on areas of busyness that I needed it to. Although comparing every single RGB value of every single colour seems like a lot, I found that it gave the most precise result in the end.

A bug that I ran into while implementing this feature was that I initially was treating the x and y pixel coordinate values as integers instead of doubles. Therefore, as I was adding values onto x and y to get subpixels (ie. 1.25 instead of 1), the values were getting truncated to ints and rounded down - essentially never changing from their original values. This made it appear like AAA was not working even though more rays were being cast.

4.3.5 Future Additions & Unexplored Areas

One addition that I think would be good for this ray tracer is to change the division of pixels from uniform division to random division within a specified area (aka jittering). This would allow for a more precise colour detecting as well as object detection. However, in order to do this, not only would a random factor need to be added, but the weighted value of each subpixel colour would need to be calculated such that it matched the distance of the cast ray from the center of its region.

Another addition could be the ability to change the number of subpixels a pixel is divided into. Currently, I have it hardcoded to divide into four for each subdivision. However, it would be interesting to see how AAA might differ when a pixel is divided by a different number.

4.4 Objective 4: Reflection

One of the first objectives I implemented for this project was reflection. This was not only because I was confident in the algorithm and idea, but because I felt (and still feel) that reflection is an essential part of any good ray tracer. Almost all materials have a combination of diffuse and specular values, and I felt that without reflection, my ray tracer would not be able to render images remotely close to what a scene should really look like.

4.4.1 Implementation

I implemented reflection using the algorithm and information that I received from lecture. However, I further conducted a bit more research in order to fully understand the concept.

For my implementation of reflective rays, I first check if an object is reflective. In other words, does its material have a specular value greater than zero? If it is, then I know that the material will reflect light and bounce it back out into the world. Thus, I need to trace these bounces to objects that it may hit. To do this, I need to trace a new reflected ray from my point of intersection towards the direction of reflection.

In this case due to the law of reflection, a reflected ray is the mirror image of the original ray pointing away from the surface of the object. This means the angle between the original ray and the surface normal is the same as the angle between the reflected ray and the surface normal.

Thus, the formula for my reflection ray is

$$reflect\_ray = \vec{V} + 2\vec{N}\cos(\theta_i) = \vec{V} - 2\vec{N}(\vec{V} \cdot \vec{N})$$

where $\vec{V}$ is the direction of the primary ray and $\vec{N}$ is the point of intersection normal.

My next step is to cast this reflection ray from the point of intersection, treating it as though it were another primary ray. This means the reflection ray will be tested against all ob- jects and will return the colour of the object it hits (or the background). Then, the colour value the reflection ray returns with it added to the current colour of the object multiplied by the object's specular factor.

It is worth noting the the object a reflection ray could hit is also reflective ($ks > 0$) which means another reflection ray needs to be cast for that object before it can return its colour to the original reflection ray. To avoid casting reflective rays for an infinite amount of time, I set a reflective ray count parameter so that only count number of reflection rays can be cast.

In order to avoid creating a reflective ray when it is unnecessary (aka material is not specular or count has hit zero), I first check to make sure a reflection ray is needed. Also, I create a base reflected colour (0,0,0) and go about my algorithm as mentioned above if reflection is needed. This allows me to always add the reflected colour to the final returned colour, even if there is no specular colour to add (in which case the added reflected colour is 0).

To combine the final reflected colour to the primary ray computed colour I chose to implement Fresnel Equations. In the section about refraction I will further explain these as I used the refractive ray to calculate them. For now it is enough to explain that Fresnel Equations balance the amount of colour being added to the primary colour from reflective and refractive rays by multiplying both reflective and refractive colours by a factor R and $(1 - R)$ respectively. If there is no refraction, then the value of R is one and the reflective colour is fully added to the primary colour (after being multiplied by the object's brightness value).

To vary the amount of reflective rays being cast, I chose to make the reflective ray count a parameter in the `a4_render` method. This way, I can set the number of reflective rays to whatever number I choose for each render I want to do. In general however, I have found that 5 reflective rays is a good value to work with.

[Baranoski, 2017c] [Blinn and Newell, 1976] [de Greve, 2006]

4.4.2 Implementation Location in Source

My implementation of reflection can be found in `A4.cpp` with a focus on the method `ray_colour`.

4.4.3 Render Examples and Demos

For some examples of reflection in action, refer to the `3 & 7 Reflection Glossy Reflection` directory in the `Final Renders` directory and search for the `glossy.png` file. Note that this render also shows off glossy reflection, which is described in a different section. I chose to render these two together as they are closely linked and glossy cannot be shown without already having reflection working.

To render an example of reflection (and glossy reflection), a LUA file can be found in Assets under the filename `glossy.lua`.

4.4.4 Challenges and Bugs

I did not face any big challenges when implementing reflection. One bug that I made sure to look out for was having my reflective rays intersect with their own originating object due to floating point error. This bug was already mentioned from A4 and sending shadow rays, so I already knew to move the point of intersection a fraction of an amount away from the object to avoid accidently intersecting it.

4.5 Objective 5: Glossy Reflection

Glossy reflection is an expansion of reflection where it gives my ray tracer the ability to render objects that have semi-reflective surfaces, such as brushed metal, where the materials reflect light, but do so in a blurred manner. Because glossy reflection is an extension of reflection, I thought it would be a good addition to my ray tracer, and after researching the technique, an objective that I could implement.

4.5.1 Implementation

To implement glossy reflection, I used a technique described in "Fundamentals of Computer Graphics, Chapter 10.11.4" by Shirley et al. where multiple reflection rays are cast from an intersection point, with each ray being randomly perturbed by some amount. The amount the ray is perturbed depends on what I will call the 'blur square'. This square is perpendicular to the original reflection ray r and has a certain width/height a. The value of a determines the degree of blurriness. In choosing a perturbed reflection ray, I sample from the blur square.

There are other techniques by which glossiness can be done, such as the two angle perturb method we discussed in lecture. However, I found that this method made the most sense to me and I found that I was able to implement it well and it produced a render I was happy with.

However, regardless of the technique used to perturb the reflected ray, the glossiness factor/blurriness of a reflective object is produced by casting multiple perturbed reflected rays instead of one unperturbed ray and averaging their final colours together to get one reflective colour. This ensures that the reflective colour of the perturbed rays is "just as strong" as one original reflective ray while introducing a level of blurriness to the original reflection colour.

Note that the more the reflected rays can be perturbed, the more objects they can hit and thus the blurrier the final reflection will be. Also, the more rays that are cast, the better the blurring effect becomes as there are more samples with different reflected colours to average from.

To implement my version of glossy, I used the brightness factor to determine how glossy/blurry a reflective material was. The larger the brightness factor, the more focused and sharper the reflection is. I also added an extra parameter to `a4_render` to determine the number of glossy rays to cast per object intersection. If the number of rays is set to zero, glossy reflection is turned off and normal reflection is used (as described in the reflection objective section). If the number of rays is greater than zero, then instead of casting one reflection ray per object ray intersection, n glossy rays are perturbed and cast out, each following their own path as a completely new reflective ray, returning their colour after hitting reflective ray count or a diffuse surface/background.

In my implementation, the 'blur square' is calculated for each object ray intersection. This is because the square needs to be perpendicular to the point of intersection. Said 'blur square' has a size a and is centered at the point of intersection. It is calculated by creating an orthonormal basis U, V , W where W vector is the original reflective ray found using the technique mentioned in the reflective ray section. U is then calculated by taking the smallest absolute value of W, setting it to zero, swapping the other two values and negating the first one. V is then calculated by determining the cross product between W and a normalized U. According to Hughes and Moller this gives an orthonormal basis. Since the original reflective ray is coming from the point of intersection and the basis is orthonormal, the other two vectors U and V will be perpendicular to the point of intersection.

Now, to create each perturbed reflective ray, I used the formula:

$$p\_reflective\_ray = original\_reflected\_ray + i * \vec{U} + j * \vec{V}$$

where $i, j = \frac{-a}{2} + rand(0,1) * a$ and $rand(0,1)$ gives a number in the range $[0,1]$.

Essentially, what the formula above is doing is taking the original reflective ray, and moving it about the 'blur square' (or finite blur plane) by a certain i amount and a certain j amount.

For my implementation, I chose to have a be determined by the brightness factor of the object's material. Thus $a = \frac{1}{1+brightness}$. Note I added +1 to the brightness value to avoid the possibility of division by zero.

In the event that the reflected ray being perturbed goes below the surface of the current object where it would count itself in its own reflection, I implemented a quick check for each of the glossy rays to avoid this issue. If the dot product of the glossy ray and the point of intersection normal is greater than zero (aka the angle between the two is larger than 90 degrees), then the glossy ray is discarded and ignored. I could have created a new glossy ray, but I did not want to run into a problem of spinning while creating new rays that kept on going below the surface.

[Baranoski, 2017a] [Shirley et al., 2005]

4.5.2 Implementation Location in Source

My implementation of glossy reflection can be found in `A4.cpp` with a focus on the method `ray_colour` and `get_glossy_ray`.

4.5.3 Render Examples and Demos

For some examples of glossy reflection in action, refer to the `3 & 7 - Reflection / Glossy Reflection` directory in the `Final Renders` directory and search for the `glossy.png` file. This render shows 5 spheres, each with a different level of glossiness.

To render an example of glossy reflection, a LUA file can be found in Assets under the filename `glossy.lua`.

4.5.4 Challenges and Bugs

One of the biggest challenges for me while implementing glossy reflection was rendering images to test this feature as well as rendering images for final renders. First, when glossy reflection is turned on, every single reflection ray that would normally be cast out now becomes n perturbed reflection rays. Thus, this increases the render time by a large factor and thus makes testing take much longer. Secondly, to get a very blurry material to look good instead of appearing speckled, a large number of glossy rays need to be cast out. Therefore, creating final renders with glossy rays took significantly longer than expected since each reflective ray was casting out about 36-64 rays each instead of the original 1.

4.5.5 Future Additions & Unexplored Areas

If I were to come back to glossy reflections and add any other details I might try to implement it using the technique mentioned in class. I think it would provide a good comparison of the two different directions and help me understand glossy reflection even more than I already do.

I would also probably change my implementation of how to handle internal reflective rays. At the moment, they are just being dropped. I think it would be a better solution to try and create a different ray and stop after a certain amount of attempts if each continue to be internal reflections.

4.6 Objective 6: Refraction

I chose to implement refraction as another one of my objectives because I felt that it was a good compliment to reflection. Not all materials with a specular component are purely reflective - some transmit or refract light as well. Thus, I felt that in order to have a robust ray tracer, I needed to implement refraction along side reflection.

4.6.1 Implementation

The general idea of refraction is a lot like reflection in the sense that a refracted ray is 'generated' at the point of intersection, cast off and traced recursively to get the colours of objects that it hits, averaging the results to get the final refractive colour.

To implement refraction, I started with the primary ray intersecting an object. My first step is to check whether the object's material is refractive. This means the object has a partially/fully transparent material and thus light goes into the object instead of being absorbed or reflected. To check this, I added a parameter to the material object called "index" which indicates the index of refraction the object's material has. This index specifies how much the material will "bend" the refractive ray before casting it. If the index is non-zero, then my ray tracer treats the material as refractive. If the index is zero, then the material is not refractive.

When a refracted ray goes from one material and enters another and the materials have different indices of refraction, then the refracted ray will bend according to Snell's Law. For the purpose of this assignment, I made the assumption that my refraction ray will always be going from air to the object material or vice versa. Therefore, storing only the object's index works since the other index is always constant.

If the material ends up being refractive, then I create a refractive ray and cast the ray out, with a direction similar to that of the primary ray. This is different than when casting a reflective ray, since the ray is "bouncing off" and the direction is opposite the original one.

To create this ray, I need to determine the angle of its bend and use the angle to calculate the direction vector. However, before I can do this I need to determine if the refracted ray is entering or leaving the object. The reason for this is because the first refracted ray will intersect with itself before any other object since it is going into itself instead of bouncing off of itself. Since the angle is dependant on entering and exiting indexes, I need to know what situation I am in.

To determine if the refraction ray is leaving an object or entering it I take the dot product of the primary ray's direction vector and the surface normal at the point of intersection. If the result is positive, the ray is exiting the object.

Otherwise it is entering.

Then, from lecture I can use the formula given to calculate my refracted ray:

$$refract\_ray = [\frac{-ni}{nt}(\vec{V} \cdot \vec{N}) - \sqrt{1 - (\frac{ni}{nt})^2(1 - (\vec{V} \cdot \vec{N})^2)}]\vec{N} + \frac{-ni}{nt}\vec{V}$$

where ni is the index of refraction of material ray is leaving and nt is the index of refraction of material the ray is entering.

The refracted ray is then cast just as the reflected ray is cast. In order to avoid an instance of infinite refractive rays, I added a count refractive ray parameter just like with reflection. I found that 5 refractive rays produced a good refractive object render.

While implementing this feature, I ran into a bug where refraction did not work when leaving an object. I was able to determine that it was because the normal was facing in the wrong direction. Therefore, while creating my refractive ray, I “flip” the normal whenever the ray is leaving the object.

Finally, I had to implement a check to avoid total internal refraction. This happens when the refractive ray is “bent” so much that instead of continuing into the object, it leaves the object, acting like a reflective ray. In other words, if the cosine of angle of refraction (transmission angle) is less than zero, then total internal refraction occurs. In this case, I simply do not cast the refractive ray.

To properly combine the reflective and refractive colours to determine the object’s final specular colour component for a given intersection point, I implemented Fresnel Equations from Hecht’s book ”Optics”. These equations determine the reflective value for s-polarized and p-polarized light and add them together to get a reflection factor R. The final specular colour is then calculated as such:

$$specular\_colour = ks * (R * reflected\_colour + (1 - R) * refractive\_colour)$$

where $R = R_s + R_p$, $Rs = (\frac{(n_1 * \cos(\theta_i) - n_2 * \cos(\theta_t))}{(n_1 * \cos(\theta_i) + n_2 * \cos(\theta_t))})^2$, and $Rp = (\frac{(n_1 * \cos(\theta_t) - n_2 * \cos(\theta_i))}{(n_1 * \cos(\theta_t) + n_2 * \cos(\theta_i))})^2$

$\cos(\theta_i)$ is the cosine of the angle of reflectance and $\cos(\theta_t)$ is the cosine of the angle of refraction. Therefore, in order to properly calculate the reflection factor R, refraction needs to be "turned on" for a given material. If it is not, then the reflection factor becomes 1.

[Baranoski, 2017c] [Blinn and Newell, 1976] [de Greve, 2006][Hecht, 2002]

4.6.2 Implementation Location in Source

Refraction is completely implemented in the `A4.cpp` file. The refracted ray is calculated using the `get_refracted_ray` method and the fresnel reflection factor is calculated using the `get_fresnel_R` method.

4.6.3 Render Examples and Demos

For some examples of refraction, refer to the `4 - Refraction` directory in the `Final Renders` directory. These renders give examples of refraction of a sphere that has different indices of refraction.

To render an example of refraction, a LUA file can be found in Assets under the filename `refract.lua`.

4.6.4 Challenges and Bugs

Out of all of the objectives in this project, refraction was the one that presented me with the most challenges. Although the concept seems simple enough, I found I had a very hard time implementing and debugging it. A huge part of the reason for this is due to the fact that the refractive ray goes through itself. This caused a lot of issues with the surface normals and determining which way it should go. I struggled with normals so much that I was only able to get refraction working for spheres and planes. In all other primitives, the refractive rays were going in directions they should not have, probably due to incorrect normals.

I also struggled with getting Fresnel to work properly. I initialy wanted to use Schlik approximations, but I could not get them to produce a correct R value. Thus I had to research more and was able to successfully implement the equations specified by Hecht in "Optics".

Finally, perhaps the biggest challenge that I faced with refraction is that I have no idea what refraction should look like for objects with varying indices. I have a general idea that refractive materials need to be "see through" but apart from that I have no idea how the items behind the object should be seen through it.

4.6.5 Future Additions & Unexplored Areas

One future addition that I could add to refraction is glossy transmission - or "foggy/semi-transparent" materials. The idea is very similar to glossy reflection. Since my ray tracer already has glossy reflection, it seems like the next step would be to add glossy transmission.

If I could, I would also spend time fixing refraction itself so that all of my primitives refract light correctly. This would probably the first thing I would do if I decided to improve this ray tracer in the future.

4.7 Objective 7: Soft Shadow

In my standard A4 ray tracer, shadows are created by sending a shadow ray from an point of intersection to a light source. However, because the light source is a point and has zero area, the shadows created are either "there" or "not there", resulting in hard shadows with distinct lines. This however, is not reflective of real shadows produced from non-zero area light sources. Instead these shadows are soft, with partially visible areas that are semi-shadowed. The reason for this is because of the penumbra, a transition zone where light goes from fully blocked to fully visible. Therefore, I decided that in order to improve my ray tracer, implementing soft shadowing would be an attainable and ideal goal to achieve.

4.7.1 Implementation

Implementation of soft shadowing follows the same concept as glossy reflection, where multiple perturbed rays are cast from a point of intersection instead of just one and the final shadow colour is determined by averaging together all of the colours returned by the perturbed shadow rays. However, instead of creating a ‘blur square’ like with glossy reflection, I created non-zero area light sources and just cast the shadow rays to point to some random place within the light.

For my project, all of my non-zero area light sources are defined as rectangular lights with one corner at point C and a defined height and width added to C to get the other three corners. I also set my lights to be rectangles on the y-plane as most of my final renders and scenes look down the z-axis with (0,1,0) as the up vector.

The original way to create a shadow ray is to create a new ray with the point of intersection as its origin and the light’s position - the point of intersection as its direction. Thus, to create a perturbed shadow ray, I changed the light’s position point to be somewhere in the light’s rectangular bounded plane using the equation below:

$$shadow\_ray = [C_x + rand(0,1) * width, C_y, C_z + rand(0,1) * height) - p]$$

where p is the point of intersection and $width, height$ are the width and height of the light.

Also, just as I did with glossy rays earlier, I added an extra parameter to the `a4_render` method to determine how many shadow rays are to be sent out per point of intersection. If the number of shadow rays is 0, then the soft shadowing is turned off and only one shadow ray is cast, using the light’s C point in the shadow ray calculation (as it was done originally). If the number of shadow rays is greater than zero, then n shadow rays are cast to somewhere in the light area, and averaged together for every point of intersection. It is important to note that unlike reflection where multiple reflection rays can be cast in sequence before a colour is determined, there is no shadow ray limit as each shadow ray either hits the light source or does not.

It is worth noting that as the number of shadow rays increases per point of intersection, the better the penumbra or “transiting section” will appear. This is because there are more rays to average and thus the shadow colour will be more conducive to surround pixels as the number of samples increases. I found that to get a non-grainy soft shadow, about 128 shadow rays need to be cast when anti-aliasing is turned off and about 32 shadow rays need to be be cast when anti-aliasing is turned on.

[Shirley et al., 2005]

4.7.2 Implementation Location in Source

My implementation of soft shadowing can be found in `A4.cpp` with a focus on the method `ray_colour`.

4.7.3 Render Examples and Demos

For some examples of soft shadowing in action, refer to the `6 - Soft Shadows` directory in the `Final Renders` directory. I have created a couple of images with different numbers of shadow rays cast to show the difference changing the number of shadow rays can make.

To render an example of soft shadowing, a LUA file can be found in Assets under the filename `soft_shad.lua`.

4.7.4 Challenges and Bugs

I did not run into many challenges when implement soft shadows. I found the process to be fairly straight forward and not too complex. However, like glossy reflection, I did struggle with render times when trying to test and render images with soft shadowing enabled. Soft shadowing takes a long time to render, even more so than glossy reflection, because every single object intersection needs to calculate a shadow ray and the number of shadow rays needs to be fairly large to produce a non-grainy shadow. To overcome this challenge, I did most of my test renders using only 8 shadow rays. This made the produced shadow very grainy, but allowed for me to have a general idea of how the feature was working and where the shadow would be in the final image.

4.7.5 Future Additions & Unexplored Areas

One future addition I would love to add to this objective is the ability to have different shaped lights as well as lights with non-zero volumes. This would allow for light sources more closely related to real life. I think it would be very interesting to be able to create a tube light or light bulb and see how it affects the shadowing of a scene.

Another future addition that could be added that is not as complex as adding volumed lights is adding the ability to change the orientation of the rectangular light. Currently, the light is always on the y-plane. It would be helpful if the light could be angled in any which way such that it could create light coming from the side or at an angle.

4.8 Objective 8: Texture Mapping

I chose to make texture mapping an objective for this project as I felt like it would bring the most "bang for its buck". My ray tracer from A4 only allowed for materials with one base diffuse colour and I felt that if I wanted a ray tracer that produced nice renders, I would need more than that.

Texture mapping is a technique in which a texture, from an image/raster file, is taken and applied to an object as its diffuse colour. It maps the point of intersection on the object to some location on the image texture file and uses the RSG colour value for that location as the diffuse colour for the point of intersection. In order to do the mapping, uv-coordinates need to be calculated for each point of intersection, taking into account the type of primitive. This uv-mapping essentially takes the point on the primitive and converts it to a 1x1 2-coordinate plane. This mapping is then converted to coordinates that match with the texture file.

4.8.1 Implementation

In order to implement texture mapping, I had to add two additional parameters to the Intersection class. These two parameters represent the u and v values for uv-mapping for a specific primitive's point of intersection. These values are then used in the rest of my implementation to calculate the diffuse colour of their corresponding Intersection objects.

Since uv-mapping is primitive dependent, I was not able to implement texture mapping for every single primitive in my ray tracer due to time constraints. I was able to implement texture mapping for planes, cubes, and spheres, each of which uses different formulas to determine their uv-mapping values.

Before I explain the implementation of each unique primitive uv-mapping, I will first explain how the uv-mapping values are used to determine the final diffuse colour of that point of intersection.

Once I get the uv-coordinates for an intersection point, I map them onto the texture map to get the diffuse colour value. Unfortunately, this cannot be done by just multiplying the u,v values by the width,height of the texture file. This is because the mapped value might fall in between a pixel of the texture file. To avoid this problem, the colour for the point u,v is deter- mined by introducing bilinear filtering. This is done by first getting the texture mapped values of u,v known as u_p and v_p, as seen below. These are calculating by subtracting the integer value of $(u, v) * (width, height)$ from the floating point value of $(u, v) * (width, height)$.

$$u_p = d_i - i$$

$$v_p = d_j - j$$

where $d_i = (width - 1) * u$, $d_j = (height - 1) * v$, $i = int(d_i)$, $j = int(d_j)$

Then, the colour for the four surrounding pixels is determined by getting the colour RGB values from the texture image from locations (i, j), $(i + 1, j)$, $(i, j + 1)$, and $(i + 1, j + 1)$. If i, j, $i + 1$, or $j + 1$ end up being larger than the texture's image size, I wrap the offending value around to the other side (ie. set it to 0).

The final colour of the point of intersection is then determined by the weighted average of the four colours computed above. The closer the point is to a pixel, the more weighted that colour is.

$$final\_colour =$$

$$colour(i, i) * (1 - u_p) * (1 - v_p) + colour(i, j + 1)(1 - u_p)(v_p)$$
$$+colour(i + 1, j)(u_p)(v_p) + colour(i + 1, j + 1)(u_p)(v_p)$$

In addition to implementing uv-mapping and thus texture mapping, I had to also add the ability to load the RGB colour values from a PNG file into my ray tracer. Thankfully, saving a PNG was already implemented and I essentially did the exact same steps except in reverse. I then used this load PNG method to load in the RGB values of the texture image files.

[Blinn and Newell, 1976] [Baranoski, 2017d]

4.8.1.1 UV-Mapping of Planes

Determining the uv-mapping values of plane object is fairly easy due to the fact that the plane is a z-plane and I made the decision to have textures map planarly according the the z-plane. Because the plane primitive is a unit plane, the uv-mapping does not have much to it. Essentially, the z value of the point of intersection is dropped to make the coordinates 2D and the x and y values are both increased by 0.5. This is because the uv-mapping values need to be in between 0 and 1, whereas the unit plane primitive is centered at (0,0,0) meaning their x and y values are in between -0.5 and 0.5.

4.8.1.2 UV-Mapping of Cubes

With uv-mapping, cubes can be mapped in two ways - planar in one direction and planar in all directions. With planar in one direction, the texture is mapped to the cube just as it would be for a z-plane, by dropping the z value and moving the x and y values. This results in a cube with its front and back faces showing the entire texture and its side showing the edges of the texture. With planar in all directions, instead of only mapping the texture onto the z-plane, the texture is mapped onto all planes, essentially mapping the texture onto all of the planes.

Since my final scene contains letter blocks, I decided that planar in all directions was a better option for this ray tracer. I also liked that it is a slightly different technique used for the plane primitive and thus would add a little more variety to my ray tracer.

The first step in uv-mapping the cube using the planar all directions method is to determine which plane the point of intersection is on. I did this by evaluating the normal of the point of intersection. Since the intersection points and normals are determined in the model view, I do not have to worry about the cube not being axis aligned. Thus, whatever x, y, z value is non-zero determines what plane the intersection is on.

If the intersection is on the z-plane, then I drop the z-value of the point of intersection and convert the x and y values. If the intersection is on the y-plane, then I drop the y value of the point of intersection and convert the x and z values. Finally, if the intersection is on the x-plane, then I drop the

x value of the point of intersection and convert the y and z values. In order to keep my mapping consistent, the first value in the x, y, z order is always converted to the u value and the second value is converted to the v value.

To convert the two points of intersection values that were not dropped, I have to take the points from the plane size and scale them down to unit size. In other words, I first subtract the cube's position from the point of intersection value to center the cube at (0,0,0) and then divide the value by the size of the cube to get a value in $[0, 1]$ instead of $[cubeposition, cubeposition + cubesize]$.

4.8.1.3 UV-Mapping of Spheres

Unlike the uv-mapping of planes and cubes, uv-mapping of a sphere is a little more complex. This is because the sphere is not some version of a plane the uv-mapping values can not be determined by just dropping one of the values. Instead, the sphere is a curved surface and thus a rectangular plane cannot fully be mapped 1-to-1 to the sphere's surface. However a good approximation can be made by using the normal of the point of intersection and specified angles. It is worth noting the planar uv-mapping can be done with a sphere, but by using the idea that the sphere, when viewed from one way looks like a disc plane. This however, results in an odd looking texture mapping and does not reflect the curvature of the sphere. Because of this I chose to not implement this method and instead to use spherical mappings instead.

The u and v values of uv-mapping can be calculated by the formulas shown below. It is important to note that the mappings can only be done if the normal is a unit vector going from sphere's centre to the point of intersection (which in my case it is) and if the sphere is axis aligned.

$$\vec{u} = 0.5 + \frac{angle1}{2 * \pi}$$

$$\vec{v} = \frac{angle2}{\pi}$$

where $angle1 = \arctan(\frac{normal_x}{-normal_z})$ and $angle2 = \arccos(\frac{-normal_y}{radius})$

[Wikipedia, 2017]

4.8.2 Implementation Location in Source

The uv-mapping implementation can be found in `Primitive.cpp`, near the end of the intersection methods. The actual mapping and diffuse colour determination can be found in `PhongMaterial.cpp` in the `get_kd()` method. Finally, the load PNG method can be found in `Image.cpp`.

4.8.3 Render Examples and Demos

For an example of texture mapping in action, refer to the `2 - Texture Mapping` directory in the `Final Renders` directory. This render gives an example of plane and cube texture mapping. To see an example of spherical texture mapping, refer to the bump mapping renders mentioned in the bump mapping section. Since bump mapping uses texture mapping uv coordinates, bump mapping a sphere is a good example to show texture mapping techniques as well.

To render an example of texture mapping, a LUA file can be found in Assets under the filename `texture_test.lua`.

4.8.4 Challenges and Bugs

One of the biggest challenges I faced with texture mapping was understanding exactly how spherical uv-mapping works. Although I was able to implement the required formulas and understand a high level explanation as to why they worked, I struggled (and still slightly struggle) with exactly why the formulas work. Even though this project has been completed, I still want to spend the time to fully understand the concept better.

4.8.5 Future Additions & Unexplored Areas

Due to the fact that uv-mapping is primitive dependent, a good extension of this objective would be to implement uv-mapping for my other primitives such as the cylinder and cone. I not only think this would make the ray tracer even more robust, but help with my understanding of uv-mapping even more.

Another addition that could be made is to have some sort of mapping type indicator. This way, a user can specify how they want the texture to be mapped to a primitive, whether its planar, spherical, cylindrical etc.

4.9 Objective 9: Bump Mapping

I chose to implement bump mapping as my final objective because not only do I feel like it is a natural extension of texture mapping, but because it allows for texture to cast appropriate shadows in a scene, even when the actual object is not that shape. This gives a better sense of realism to a final render and I think it adds the "final polish" to my ray tracer.

Just as texture mapping maps an image to an object, bump mapping maps a literal texture to an object, making it appear as though the object has a non-uniform surface (aka non-smooth) by displacing the surface normal certain points on the object. The reason bump mapping is a useful technique is because creating an object with a non-uniform surface is extremely difficult to do due to intersection calculating. By "pretending" the surface is non-uniform after finding an intersection, small details can be added without changing the simpler basic primitive intersection methods.

4.9.1 Implementation

Bump mapping is very similar to texture mapping in that it uses the same uv-mapping techniques. However, instead of grabbing the colour from the texture file, comparisons are done in the bump file to determine a certain displacement amount to apply to the surface normal of the object. Therefore, I used the same uv-mapping techniques that I implemented for each texture mapped enabled primitive.

Since the surface normal of a point of intersection is going to be displaced, I need to add two more values to my Intersection class. These values, known as $\vec{O_v}$ and $\vec{O_u}$ are the partial derivatives of the surface described by the uv-mapping u and v values. These partial derivatives are used later on in the bump mapping calculation to appropriately perturb the normal. These partial derivatives (which are vectors) are used because they are tangent to the surface of the object at the current point of intersection. Just as uv-mapping is specific to each primitive, determining $\vec{O_v}$ and $\vec{O_u}$ also require unique formulas per primitive. However, before explaining each calculation, I will continue to explain how these derivative are used to disturb the intersection normal.

Instead of using bilinear interpolation to get the colour from the image file as I did with texture mapping, I will be using the sampled bump value to displace the surface normal of my object to give the illusion of bumps on the surface. The normal of the object's surface is displaced by adding a displacement value to it. In other words, $\vec{N'} = \vec{N} + \vec{D}$. This displacement value, $\vec{D}$, is calculated using the partial derivatives of the surface as well as the displaced amount from the bump mapping.

Essentially,

$$D = B_u * \vec{X} - B_v * \vec{Y}$$

where B_u and B_v are partial derivatives of the bump map at u, v, $\vec{X} = \vec{N} \times \vec{O_v}$ and $\vec{Y} = \vec{N} \times \vec{O_u}$

To calculate B_u and B_v, I used two equations given to us in lecture. Basically, what these equations do is sample the colour value of the bump at four different locations e distance from the actual point. They then use this colour difference to extrapolate the tangent lines or partial derivatives at each point and are then averaged together using weights to return a final partial derivative amount.

$$B_u = \frac{B(u+E,v) - B(u-E,v)}{2 * E}$$

$$B_v = \frac{B(u,v+E) - B(u,v-E)}{2 * E}$$

where $E = \frac{1}{512}$ (which I determined after playing around with different values) and $B(u, v)$ is a function that samples the bumpmap file at the bump coordinate given using bilinear filtering. This function $B(u, v)$ uses the exact same steps as texture mapping but instead of returning a RGB colour value, it returns just the R colour value.

I chose to only sample the R value of the bump map because I made the decision to only provide grayscaled bump maps. This way I did not have to determine a more complex way of determining colour difference.

I also had to include a scaling factor to the final displacement vector. I did this because I found that the normal was being displaced too much for my liking, making the final bump mappings and renders very shadowed and harsh. I played around with a number of different values before deciding on 0.05.

The final displaced normal is then used as the new normal to calculate lighting calculations such as shadows.

[Blinn, 1978] [Watt, 1992] [Baranoski, 2017b]

4.9.1.1 $\vec{O_u}$ and $\vec{O_v}$ Implementation for Plane

Due to the fact that the plane primitive is in fact a plane, finding the partial derivatives for the point of intersection was fairly easy. This is because the partial derivative, or the tangents to the point are just vectors on the plane. Since my plane primitive is a unit z-plane, all I had to do was set $\vec{O_v}$ and $\vec{O_u}$ to the two vectors opposite the normal of the plane (0,0,1). Thus,

$$\vec{O_v} = (0, 1, 0)$$

$$\vec{O_u} = (1, 0, 0)$$

4.9.1.2 $\vec{O_u}$ and $\vec{O_v}$ Implementation for Box/Cube

Just as I did for texture mapping, I wanted to do planar in all direction bump mapping on the cube instead of plane in one direction bump mapping. This allowed me to reuse the u and v values as well as match my bump mapping to a corresponding texture.

However, unlike my plane primitive, I cannot just set the $\vec{O_v}$ and $\vec{O_u}$ vectors to be the other two e_i vectors to the plane normal because my cube is not always a unit cube. Therefore, I have to do a little more work.

Essentially, $\vec{O_v}$ and $\vec{O_u}$ will be the other two vectors in a 3 vector orthonormal basis including the surface normal. Because the normal is perpendicular with the plane that the point of intersection is on, the other two vectors in its orthonormal basis will be two vectors on the plane, just as needed.

Just as I did with glossy reflection when creating a 'blur square' I will use the technique mentioned by Hughes and Moller to create the orthonormal basis using the normal as the starting vector.

[Hughes and Moller, 1999]

4.9.1.3 $\vec{O_u}$ and $\vec{O_v}$ Implementation for Sphere

Unsurprisingly, calculating the $\vec{O_u}$ and $\vec{O_v}$ vectors for a sphere is much different than for a plane or a cube. This is because the partial derivatives are no longer just vectors along the surface of the primitive's surface, but instead actual tangents to the surface at the point of intersection. Luckily, there exists formulas that calculate the partial derivatives of a point on the surface of a sphere using the same angles used in the uv-mapping step of texture mapping.

Therefore,

$$\vec{O_u} = [-r * \sin(angle1) * \sin(angle2), 0, -r * cos(angle1) * sin(angle2)]$$

And

$\vec{O_v} = [r{*}\cos(angle2){*}\cos(angle2), r{*}\sin(angle2), -r{*}\sin(angle1){*}\cos(angle2)]$

where r = radius of the sphere

4.9.2 Implementation Location in Source

The implementation of bump mapping can be found in the same files as texture mapping. The $\vec{O_u}$ and $\vec{O_v}$ implementation can be found in `Primitive.cpp` in the intersection methods. The displacing of the normal can be found in `PhongMaterial.cpp` in the `get_bumped_normal` method. Finally, the loading of the bump image file can be found in `Image.cpp` in the `loadPNG` method.

4.9.3 Render Examples and Demos

For some examples of bump mapping in action, refer to the `8 - Bump Mapping` directory in the `Final Renders` directory. These renders give examples of planar, cubic, and spherical bump mapping, each with the light in a different place to demonstrate the location of the shadow according to the bumped normal. I also included a GIF of the renders to see how the shadows move as the light moves.

To render an example of bump mapping, a LUA file can be found in Assets under the filename `bump.lua`.

4.9.4 Challenges and Bugs

Just as with texture mapping, my biggest challenge with bump mapping was understanding why $\vec{O_u}$ and $\vec{O_v}$ can be calculated the way that they are for a sphere. However, unlike texture mapping, one reason that I had difficulty with understanding this is because I have not had instruction in partial derivatives. I believe that once I understand those better, than reasoning out the partial derivatives of a sphere should be more straight forward for myself.

4.9.5 Future Additions & Unexplored Areas

Since bump mapping is so intertwined with texture mapping, any future additions that I would want to add to texture mapping, I would also like to implement for bump mapping. This means that regardless of the texture, my bump mapping would be able to accommodate it. I believe this would keep the project consistent and polished.

Another possible future addition would be to actually perturb the surface instead of the normal. This is known as displacement mapping and allows for self-shadowing, something bump mapping does not do.

4.10 Objective 10: Final Scene

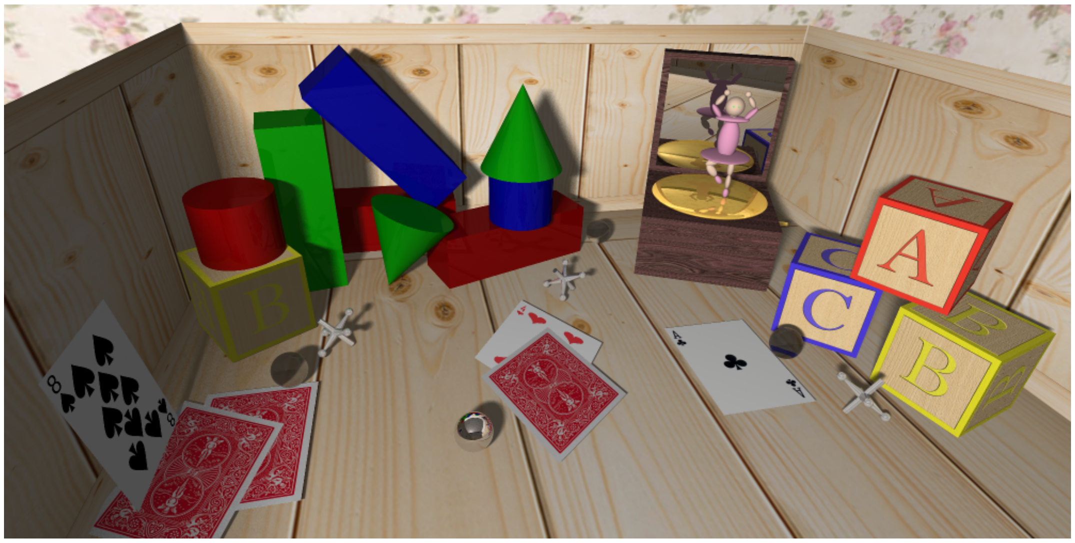
Figure 1: Final Scene - A Look into a Toy Box

5.0 Acknowledgements

As this project was an extension of A4, the project was built around basic code provided for said assignment.

6.0 References

Baranoski, 2017a Baranoski, G. V. G. (2017a). Introduction to computer graphics. University Lecture.

Baranoski, 2017b Baranoski, G. V. G. (2017b). Introduction to computer graphics: Bump mapping. University Lecture.

Baranoski, 2017c Baranoski, G. V. G. (2017c). Introduction to computer graphics: Reflection and refraction. University Lecture.

Baranoski, 2017d Baranoski, G. V. G. (2017d). Introduction to computer graphics: Texture mapping. University Lecture.

Blinn, 1978 Blinn, J. F. (1978). Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):286–292.

Blinn and Newell, 1976 Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547.

de Greve, 2006 de Greve, B. (2006). Reflections and refractions in ray tracing.

Drakos, 1997 Drakos, N. (1997). Ray tracing primitives. `https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html`.

Glassner, 1989 Glassner, A. S. (1989). *An Introduction to Ray Tracing*. Elsevier.

Hecht, 2002 Hecht, E. (2002). *Optics*. Pearson Education, Inc, 4th edition.

Hughes and Moller, 1999 Hughes, J. F. and Moller, T. (1999). Building an orthonormal basis from a unit vector. *Journal of Graphics Tools*, 4(4):33–35.

Mitchell, 1990 Mitchell, D. P. (1990). The antialiasing problem in ray tracing. *SIGGRAPH 1990 Course Notes*.

Roth, 1982 Roth, S. D. (1982). Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109 – 144.

Shirley et al., 2005 Shirley, P., Ashikhmin, M., and Marschner, S. (2005). *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition.

Watt, 1992 Watt, A. H. (1992). *Advanced Animation and Rendering Techniques: Theory and Practise*. ACM Press.

Wikipedia, 2017 Wikipedia (2017). Uv mapping. `https://en.wikipedia.org/w/index.php?title=UV\_mapping\&oldid=771082244`.