Ray Tracer Revised Proposal

Name: Sarah Fraser
Student ID: 20458408
User ID: sm2frase

Project Proposal for Final Project:

Purpose :

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

Statement :

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

For each objective I plan to research techniques and algorithms on how the problem is solved and implemented. I will then attempt to add each objective to my ray tracer. I will then test each objective using a basic modelling scene that I have created/found that highlights the given objective. For example, my reflective objective will be tested against a scene containing an object with a highly reflective material. Finally, I plan to apply all of my techniques to my final scene to fully show what my ray tracer is capable of doing.

If I have extra time I also plan to implement other objectives that I find interesting. Some possible additions include depth of field, CSG, and 3D stereo vision. I am particularly interested in the stereo vision.

On top of my objectives and final modelling scene, I plan increasing the efficiency of my ray tracer using multithreading techniques. This will allow me to render scenes faster and thus help with my time management.

Finally, I plan on creating a basic webpage that displays and explains each of my 10 objectives and their test scenes. The webpage will also display my final scene and any other extra objectives I may finish.

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.

I believe that the most challenging part for me will be managing my time. Although I find the content very interesting, it takes me a while to fully grasp each concept. I am worried that I will not have enough time to implement everything because I spent too much time trying to understand it. Also, I believe that rendering time will be another challenge I have to face. Anti-aliasing, soft shadows, and glossy reflection are all objectives that require me to cast additional rays per pixel (anti-aliasing from the pixel and glossy/soft shadows from an object). This will increase my render time. Hopefully, by implementing multi-threading I can combat this issue and reduce my rendering time by some amount.

By the end of this project I hope that I have learned a vast amount about ray tracing and the techniques used to achieve it. I also hope that this project allows for me to improve my ability to implement algorithms I have read from papers and books so that I can do so in future situations in my life. Finally, I hope that by the end of this project I will have learned how exciting graphics and especially ray tracing, can really be. I have only just begun researching this project and I am already excited to get started implementing it.

Final Scene :

For my final scene, I plan to implement a “left behind toys” scene. This scene will have a toy box object (made from a cube and cylinder) with a wooden texture applied. Around the tox box will be various types of toys. These will include marbles, balls, dice, jacks, cards, and building blocks of various shapes. All of the objects will be sitting on a plane (with a table texture) and have a background

of a wall. The various toys will have different materials. Some, such as the marbles, will be transparent, others will be glossy, some reflective, some matte. Some of the toys will have mapped textures (such as the cards) while others will have bump mappings (such as the dice). There will be a light above the scene, which will produce shadows of the box and toys onto the plane table they are sitting on.

To achieve this scene I will need to implement all 10 of my objectives. This will allow me to create all of my desired shapes and material effects, as well as make it look fairly realistic.

Technical Outline :

- Objective 1: Primitives

  For this objective, I plan to add two additional primitives to my ray tracer from A4. These two primitives are the cylinder and the cone. I chose these two primitives as they both present more of a challenge than just implementing a primitive with all planar faces (ie. a cube, icosahedron). Instead, both have a combination of planar faces and curved surfaces. Therefore, instead of just checking for an intersection of each face and then determine inside/outside (as I do for cube already) I need to do additional work and research.

  I am planning to have both primitives to be unit shapes and extend about the z-axis. That is, both curves go 'around' the z-axis. When implementing the new primitives, both will need a new function in the Lua interpreter and new primitive classes within the ray tracer program. New intersection functions will be needed to be defined as well.

  The cylinder function in the Lua interpreter will look as follows: **gr.cylinder(name)**
  This function will create an axis-aligned cylinder with one end centred at (0,0,0) and the other (0, 0, 1).

  The cone function in the Lua interpreter will look as follows: **gr.cone(name)**
  This function will create an axis-aligned cone with the point centered at (0,0,0) and the 'bottom' face centered at (0, 0, 1).

  I will now go into a small amount of detail on how each primitive's intersection check will work. First, starting with cylinder. A cylinder has two faces and a curved side. Thus, an intersection needs to be checked for all three. I will start with the curved side, using the formula for an infinite cylinder ($X^2 + Y^2 = R^2$ where $R = 1$ in this case). Then, I will solve for t using the same technique as a sphere. I will then give bounds to the Z values of the points of intersection since I have a *finite* cylinder. Depending on the number of solutions I can have 3 situations:

  1) No solution and thus no intersection with the curves surface.
  2) One solution meaning the ray is tangent to the curved surface. If the Z value of the point of intersection is within the bounds then the point is valid.
  3) Two solutions meaning the ray intersects the curved surface. If at least one of the Z values of the points of intersection are within the bounds, the point(s) are valid.

  When there are two solutions and one Z value falls outside of the bounds, this means the ray intersects the cylinder through one of its faces and the curved side. A third t value needs to be found to find the point of intersection with the face.

The final point of intersection is the point that a) has a Z value within the bounds OR hits the face and b) has the smallest t value of the remaining points from the previous specification.
Finally, if no intersection points are found with the curve, a test for intersection with the just the faces needs to be done. This is a simple plane intersection test.

The cone intersection test is very similar to the cylinder test, except for the equation for an infinite cone is $X^2 + Y^2 = Z^2$, and there is only one face instead of two. There also does not require a face only intersection test as there is no possible way for the ray to only intersect the face and not the curved surface as well.

[1] [2] [3]

I plan to show this objective by rendering a scene with these two primitives.

- Objective 2: Animation

To implement animation, my plan is to rotate my scene objects about the y-axis for a 360 view of the object (aka one full rotation). I plan on doing this by rotating the root node around the y-axis of my scene by an n number of degrees per frame. It should look as though my scene is being rotated on a disk (like in a microwave). Because I am not planning on rotating the light source(s), the shadows should not rotate with the objects and instead should stay relative to the objects' positions with the light source(s).

To create my animation, I need to render a set number of frames. Thus, I need to determine how much to increase the angle of rotation per frame of my animation and how long I want my animation to be.

Recall that in order to get a smooth looking animation, I will need 30 frames/second.

- If I rotate by 1 degree every frame, then that is 360 frames in total. This will result in a full rotation done in 12 seconds. I feel like this is far too long for just a simple rotation.
- If I rotate by 2 degrees every frame, then that is 180 frames in total. This will result in a full 6 seconds of animation. I still feel like this is too long of an animation.
- If I rotate by 3 degree every frame, then that is 120 frames in total. This is will result in a 4 second animation. This seems like a reasonable animation time to show a rotation. This is also a reasonable amount of frames to create a small .gif that will loop my animation every time.

[4]

I will be doing my animation on a scene that consists of one or two basic primitives that do not require long times to render. This will help me save time when creating all of my rendered image frames for the animation.

- Objective 3: Adaptive Anti-Aliasing

To implement anti-aliasing, I am choosing to use an adaptive approach (ie. only performing anti-aliasing techniques on 'busy' areas of the image) instead of a uniform one. The reason for this is because I would like to improve my efficiency when rendering and one of the best ways to do that is to minimize the number of rays cast. By performing anti-aliasing techniques on every pixel, one wastes resources on pixels that have little change in colour to them, such as pixels showing a

background or ones in the middle of a solid coloured object with no shadow/reflection variance. Casting more rays for those pixels will return close to/the same results as the previous rays, thus wasting resources and time since no improvement can be made.

The adaptive anti-aliasing technique I would like to implement is one described by Don P. Mitchell in his paper 'The Anti-Aliasing Problem in Ray Tracing'. It is worth noting that Mitchell himself is expanding upon/explaining clearly the adaptive anti-aliasing technique described by Whitted.

The technique is described as follows:

- For each pixel, cast a ray for each corner of the pixel.
- 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 size subpixels.
- 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.
- 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.
- 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.

[5]

It is worth noting that while completing A4, I implemented supersampling by casting 4 rays from each corner and averaging them. This resulted in a picture that looked blurry as the colours were being shared between pixels. To fix this, I cast 4 rays from the centre of four divisions of the pixel. Because of this, I plan on testing where I should cast the 4 rays from my pixel and if moving them from the corners to the centre will make any difference.

The threshold and maximum number of divisions are numbers that I need to determine by trial and error. However, I will most likely start by setting my maximum number of divisions such that the weighted average of the corners of a subpixel does not go below 1.

I plan to show this objective by rendering three images of the same basic scene (with obvious jagged lines):

- The first render will be without adaptive anti-aliasing
- The second render will shade the pixels that will be anti-aliased using a bright colour (ie. red)
- The third render will be with adaptive anti-aliasing turned on

I will add an option in the Makefile such that one can turn anti-aliasing on and off.

- Objective 4: Reflection

I plan to implement reflection by using the technique shown to the class by Professor Baranoski. I will give a basic overview of the technique as follows.

I first need check if an object is reflective. In other words, does its material have a specular value greater than zero? If it does, 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 $\vec{reflectray} = \vec{ray} - 2\vec{N}(\vec{V} \cdot \vec{N})$.

The next step is to cast this reflection ray from the point of intersection, treating it as though it were a ray coming from the eye. This means the reflection ray will be tested against all objects and will return the colour of the object it hits. Then, the colour value the reflection ray returns with is 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 infinity, a maxhits parameter needs to be set so that only maxhits number of reflection rays can be cast.

[6] [7] [8]

To implement this objective, I will need to change my rayColour function to include casting reflection rays.

I plan to show this objective by creating a scene where multiple different primitives have a reflective surface, with some having a greater specular factor than others. I will also include primitives that do not have a reflective material to add variance in the reflected colour values.

- Objective 5: Refraction

Just as I plan to implement reflection using information lecture, I also plan to implement refraction in the same way, using what was given to me by Professor Baranoski. The technique for refraction is as follows.

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 and sum these colours to get the final colour of the original object. Also, like reflection, I will implement refracted rays to have a maxhit limit so that I am not casting refraction rays for an infinite amount of time.

Just like reflection, once my eye ray intersects an object, I first need to check if said object 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. Eventually, the light will then exit the object and continue on to hit other objects. Thus, when the eye ray hits a refractive material, the refracted ray will intersect the object twice - once going into it and once going out of it.

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. Thus, I need to account for this. For the purpose of this assignment, I am going to make the assumption that my refraction ray will always be going from air to the object material or vice versa. Therefore, I only need to store the index of refraction for the object material since air's index is 1 and will stay constant.

To calculate the refracted ray and its angle of bend, I need to first determine if it is leaving an object or entering it. This is easily done by taking the dot product of the ray's direction vector

and the surface normal. If the result is positive, the ray is exiting the object. Else entering.

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

$$\vec{refractray} = [-\frac{n_i}{n_t}(\vec{V} \cdot \vec{N}) - \sqrt{1 - (\frac{n_i}{n_t})^2(1 - (\vec{V} \cdot \vec{N})^2)}]\vec{N} + -\frac{n_i}{n_t}\vec{V}$$

where n_i is the index of refraction of material ray is leaving and n_t is the index of refraction of material the ray is entering.

The refracted ray is then cast just as the reflected ray is cast.

Upon further research on reflection and refraction, I looked further into the concept of Fresnel equations. Essentially, Fresnel equations determine how much light is transmitted (refracted) and how much light is reflected. Fresnel equations can be approximated using Schlick's Approximation. Thus to determine how much of my eye ray is reflected and how much of my eye ray is refracted (aka how much of my final object colour is determined by each ray) I will use the approximation below. /newline

$R_0 = \frac{(n-1)^2}{(n+1)^2}$

$R = R_0 + (1 - R_0)(1 - \vec{refractray} \cdot \vec{N})^5$

$colour = ks(R * rayColour(\vec{reflectray}) + (1 - R) * rayColour(\vec{refractray}))$

where n is the index of refraction for the material the ray is leaving.

[6] [7] [8]

To implement refraction, I will need to add another parameter to the material object in the Lua interpreter. This parameter will be the index of refraction for a material. I will also need to change my rayColour function to include casting refraction rays. It might also be beneficial to create a new function that computes the refraction ray.

I plan to show this objective by creating a scene where multiple different primitives have a refractive surface (indicated by an index of refraction ¿ 0), with some having a greater index of refraction than others. I will also include primitives that do not have a refractive material to add variance in the colour values.

- Objective 6: Glossy Reflection

To implement glossy reflection, I am going to use a technique 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 'blurr square'. This square is perpendicular to the original reflection ray r and has a certain width a. The value of a determines the degree of blurriness. To choose a perturbed reflection ray, I will sample from the blurr square. The blurr square, which has size a and is centered at the point of intersection, is calculated by creating an orthonormal basis $\vec{U}, \vec{V}, \vec{W}$ where the $\vec{W}$ vector is the original ray. $\vec{U}$ can be calculated by taking the smallest absolute value of $\vec{W}$, setting it to zero, swapping the other two values and negating the first one. $\vec{V}$ can be found by calculating the cross product between $\vec{W}$ and a normalized $\vec{U}$. According to Hughes and Moller [9] this gives an orthonormal basis.

Then, to create the perturbed ray, I will use the formula:

$\vec{r'} = \vec{r} + i\vec{U} + j\vec{V}$

where $i, j = -\frac{a}{2} + rand() * a$

The random number generated to calculate i, j will be a number between $[0, 1]$.
For my implementation, I will be setting a to be determined by the specular factor of the object's material. In other words, $a = \frac{1}{1+ks}$. This way, I do not need to add another parameter for the glossiness of a material, and the glossiness will stay proportional to the specular factor.

To determine the number of perturbed reflection rays I am going to cast from the intersection point, I will use a variable called *glossy_count*. The larger *glossy_count*, the more rays will be cast per intersection.

To determine the final reflection colour, the reflected rays' colours are summed together and then divided by the number of rays cast to average it out. The ensures that the reflective colour of the perturbed rays is not as strong as the original reflective ray and instead adds to it to introduce a level of blurriness to the original reflection.

In my research sources, it is noted that I need to implement a test to check if any of the perturbed reflected rays go below the surface of the object. If so, I will essentially ignore that ray as the colour value I will get from it will be from inside the object and thus not a reflection I want to express.

[4] [10]

Since I am implementing reflection and glossy reflection, I will include a switch to turn glossy reflection on and off in order to render scenes with/without the objective.

Like reflection and refraction, I plan to show this objective by creating a scene where multiple different primitives have a reflective surface, with some having a different levels of reflectiveness. I will then render the scene with a fairly size *glossy_count*. I will also include primitives that do not have a reflective material to add variance in the colour values.

- Objective 7: Soft Shadowing

Like glossy reflection, soft shadowing makes use of multiple perturbed rays cast from the point of intersection. However, this time it is the shadow rays instead of the reflection rays.

To implement soft shadowing, I need to first introduce the concept of having a shaped light instead of a single source of light. This is because in real life, lights have a non-zero area. This leads to shadows where partial light 'gets through' (called the penumbra). This however is not possible if the light source comes from a single point, as it does currently in my A4 ray tracer. This penumbra is what gives me the soft shadow instead of a stark, harsh line shadow I have now. For this projection, I have decided to implement a shaped light in the shape of a rectangle defined by a point C (its corner) and H, and W (the height and width of the light).

To create this new light, I will be adding a new light type to the Lua interpreter. This new light

type will be as follows:

$gr\_rect\_light(position, colour, attenuation, width, height)$

Now, to implement soft shadow, multiple shadow rays are cast from the point of intersection to a random spot on the shaped light. The resulting shadow colour for the pixel is the average of the shadow rays' colours.

To find a point on the shaped light to cast a ray to, I will be using this formula:

$lightpoint = C + rand() * H + rand() * W$

where the random generated numbers fall in the $[0, 1]$ range.

I will then send my shadow ray to this point, calculating the shadow colour just as I would if the light was a single point in the scene.

[10]

Just as I have for glossy reflections, the number of shadow rays that I cast will depend on a variable called *shadow_count*. The larger *shadow_count* is, the more shadow rays will be cast. I will set up my Makefile so that the value of *shadow_count* can easily be changed.
It is worth noting that the more shadow rays are cast, the better the soft shadow will end up looking. I plan to experiment with different numbers of shadow rays to determine what the optimal number will be such that I can achieve a realistic soft shadow without drastically slowing down my render time.

I plan to show this objective by creating a simple scene that includes an object above another and a shaped light. This shaped light should cast a soft shadow of the first object onto the second one. I also plan on rendering a couple scenes with different values set for $shadow_count$ to show the difference increasing/decreasing the number of shadow rays can make.

- Objective 8: Texture Mapping

  Texture mapping is a technique in which a texture, from a raster file, is taken and applied to an object as its diffuse colour. For my project, I plan to implement it by generating uv-texture coordinates for each of my primitives on an intersection point. I will then take these coordinates and use them to index onto the given texture map to give me the correct diffuse colour for that intersection point.

  Since I need to use files to map the textures, I will need to update the Lua interpreter to allow for a geometryNode to have a texture file from which the texture information can be pulled from. To add a texture to a geometryNode, I will add the function **set_texture(filename)**. This function will then add the texture to a field of the PhongMaterial class associated with the given geometryNode.

  Since I need to calculate the uv coordinates (which are in the range [0,1]) for each of my primitives, my plan is to start with one mapping technique (planar mapping) and add techniques that pertain to certain primitives as I progress. This way, I will at the minimum have a way to map all

primitives, even if it is not the most accurate mapping. For example, I will planar map a texture to a sphere primitive until I do a sphere specific mapping which will wrap the texture onto the sphere instead of flatten on top.

The three main types of uv mapping that I want to accomplish are planar, spherical, and cylindrical, however I will focus on planar first and add the others if time is available.

To implement my basic planar mapping technique I first need to take the point of intersection from world coordinates to the model coordinates. This will set the point of intersection to have coordinates in the range [0,1] (which is needed for uv-mapping). Then, I will simply drop the Z coordinate of the point of intersection to get my u,v coordinates. This will set all textures to map ontop objects as if the texture was laid onto the side of the objects.

Once I get the uv-coordinates for an intersection point, I will have to 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 determined by introducing bilinear filtering. This is done by first getting the texture mapped values of u,v. Then, the colour for the four surrounding pixels is determined. The final colour is then the weighted average of those four colours. The closer the point is to a pixel, the more weighted that colour is.

[7] [11]

I plan to show this objective by creating a modelling scene such that I have different primitives displaying different textures from texture files given to them in the .lua scene file. This will allow me to display how my texture mapping looks on a variety of different primitives.

- Objective 9: Bump Mapping

  Like texture mapping, this rendering technique uses a texture mapping to give texture to an object in my scene. However, instead of grabbing the colour from the texture raster file, comparisons are done to give a displacement amount applied to the surface normal of the object. This gives the illusion that the surface of the object has real texture or bumps.

  Since I need to use files to map the bump textures, I will need to update the Lua interpreter to allow for a geometryNode to have a bumpmap file from which the texture bump information can be pulled from. To add a bumpmap to a geometryNode, I will add the function **set_bumpmap(filename)**. This function will then add the bumpmap to a field of the PhongMaterial class associated with the given geometryNode.

  Since I am still mapping from the scene to a texture file, I need to implement uv-mapping again. I will use the same methods as seen in texture mapping above to get my u,v coordinates for my object. I will also use the same methods to sample a value from the texture map using bilinear filtering. However, the value I will be sampling will be different.

  Now, instead of using bilinear interpolation to get the colour from the texture, I will be using the sampled 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 then determined by $\vec{D} = \vec{B_u}\vec{X} - \vec{B_v}\vec{Y}$ where

$\vec{B_u}$ and $\vec{B_v}$ are partial derivatives of the bump map at u, v, $\vec{X} = \vec{N} \times \vec{O_v}$ and $\vec{Y} = \vec{N} \times \vec{O_u}$ where $\vec{O_v}$ and $\vec{O_u}$ are tangent vectors of the surface of the object (ie. partial derivatives). Therefore, to calculate the amount of displacement, I need the tangent vectors for the surface of the object at the intersection point as well as the partial derivatives of the texture mapping corresponding to point u,v.

Since I have different primitives, calculating the tangent vectors $\vec{O_v}$ and $\vec{O_u}$ will depend on the type of primitive. Therefore, I plan on implementing bump mapping for only a small subset of primitives for which I know how to obtain the tangent vectors fairly well. I plan on implementing bump mapping for the sphere and cube primitives. If I have additional time I will add other primitives.

Then, $\vec{B_u}$ and $\vec{B_v}$ can be found using these equations given to us in class.

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

where $E = \frac{1}{64}$ (however, in can be another value) and B(u,v) is a function that samples the bumpmap file at the texture coordinate given using bilinear filtering.

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

[12] [13] [14]

I plan on showing this objective by creating a modelling scene where one or two supported primitives display a bump mapping given to them in the .lua scene file. It is important to note that I will not be adding textures to this scene so that I can accurately display bump mapping instead of just mapping what appears to be a bumpy texture.

- Objective 10: Final Scene

  See my description earlier for information on the final scene.

Bibliography :

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

[2] Andrew S. Glassner. *An Introduction to Ray Tracing*. Elsevier, 1989.

[3] Nikos Drakos. Ray tracing primitives, 1997.

[4] Gladimir V. G. Baranoski. Introduction to computer graphics. University Lecture, 2017.

[5] Don P Mitchell. The antialiasing problem in ray tracing. *SIGGRAPH 1990 Course Notes*.

[6] Gladimir V. G. Baranoski. Introduction to computer graphics: Reflection and refraction. University Lecture, 2017.

[7] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[8] Bram de Greve. Reflections and refractions in ray tracing, 2006.

[9] John F Hughes and Tomas Moller. Building an orthonormal basis from a unit vector. *Journal of Graphics Tools*, 4(4):33–35, 1999.

[10] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2005.

[11] Gladimir V. G. Baranoski. Introduction to computer graphics: Texture mapping. University Lecture, 2017.

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

[13] Alan H. Watt. *Advanced Animation and Rendering Techniques: Theory and Practise*. ACM Press, 1992.

[14] Gladimir V. G. Baranoski. Introduction to computer graphics: Bump mapping. University Lecture, 2017.

Objectives:

Full UserID:_____ **Student ID:**_____

___ 1: Primitives. I will be adding a cone primitive and a cylinder primitive.

___ 2: Texture Mapping. The mapping will depend on the type of primitive. However, for most primitives, planar mapping will be used.

___ 3: Reflection.

___ 4: Refraction.

___ 5: Adaptive Anti-Aliasing.

___ 6: Soft Shadows.

___ 7: Glossy Reflection.

___ 8: Bump Mapping.

___ 9: Animation. I will be doing a camera full rotatation for my animation.

___ 10: Final Scene. See statement for a description of the final scene

A4 extra objective: Simple super-sampling where every pixel was divided into four subpixels, each casting a ray used to average the final pixel colour.