Ray Tracing: Blobs

Lingyun Guo
l35guo
20586076

April 4, 2019

Contents

List of Figures

1 Statement

Ray tracing is a rendering technique where we trace light as it travels in the scene and compute colour for each pixel in the generated image. By following the path of lights or rays, we are able to simulate how our eyes' perception of lights and generate realistic images.

There are a couple advantages of ray tracing over other rendering techniques. The most notable one is that this technique traces lights just like how our eyes follow rays to perceive objects around us, which means that many physical phenomena in our daily life, such as reflection and refraction, can be easily achieved using ray tracing to produce a high level of realism. Besides that, since each pixel in image is compute separately, ray tracing can easily be adapted to parallelization, which can greatly subsidize its performance.

The basic idea of ray tracing is that, for each pixel on the image, we shoot a ray from the eye, and follow that ray to see if it intersects with any objects in the scene. If we find such an object, we execute shading model to determine what colour should be seen at that intersection, and collect the colour value to put into our image.

On top of the basic process, we can add additional features to make the image look more realisitc. For example, supersampling can be used in the ray shooting process to reduce aliasing, produce motion blur, and create effects such as soft shadow and glossy reflection.

In this project, I implemented a simple ray tracer with several additional features, and used it to generate an interesting final scene. In the final scene I created two little creatures, which I got the idea from the Blob emoji that was trending on Android platform a couple years ago. The final scene also includes a moon that is generated using Perlin noise, everything is contained in a room with light set on the top.

2 Technical Outline

2.1 Extra Primitives

In this project I implemented the following primitives: sphere, cube, plane, torus, cylinder and the shape of blobs. All of them inherit from the Primitive class, and have a member method that can compute its intersection with a ray in its model coordinate. For each of these primitives, we transform a ray from world coordinate into its model coordinate, plug the parametric equation for ray in the implicit equation defined in each class, and solve for roots that satisfy all requirements.

For the purpose of making the following discussion easier, we define a ray as $O + t \times D$, where O is the origin and D is the direction.

2.1.1 Cylinder

A cylinder can be considered as a stack of infinite circles, the computation of its intersection with rays is thus straightforward. Assuming a standard cylinder is centered at $(0,0,0)$ and has its radius and height both equal 1, then we can first solve the following equation

$$(x^2 + z^2) = 1$$
$$\Rightarrow (O_x + t \times D_x)^2 + (O_z + t \times D_z)^2 = 1$$

Once we find a valid value for t, we test if $O_y + t \times D_y$ is in between $[0,1]$, which gives us the cylinder sits at $(0,0,0)$ and has a height equal to 1.

2.1.2 Torus

The shape of a torus can be defined by 2 variables, the distance from the center of the tube to the center of the whole torus R, and the radius of the tube r. Consider a torus lies on the (x,z) plane with center at $(0,0,0)$, if we only consider the (x,y) plane, we can see that each point (x,y) on the surface of torus satisfies the following equation:

$$(x - R)^2 + y^2 = r^2$$

Note that on (x,y) plane, $z = 0$. Now if we extend this to all the plane that is aligned with y axis and goes through $(0,0,0)$, then we have:

$$x^2 + z^2 = d^2$$

$$(d-R)^2+y^2=r^2$$

Replace (x,y,z) with $(O_x+t\times D_x, O_y+t\times D_y, O_z+t\times D_z)$, and R with 1, we get the final quartic equation:

$$\begin{aligned}(D\cdot D)^2\times t^4+\\ 4\times D\cdot O\times t^3+\\ (2\times D\cdot D\times (O\cdot O-(r\times r+1))+4\times D\cdot O\times D\cdot O+4\times D_y\times D_y)\times t^2+\\ (4\times (O\cdot O-(r\times r+1))\times D\cdot O+8\times O_y\times D_y)\times t+\\ (O\cdot O-(r\times r+1))^2-4\times (r\times r-O_y\times O_y)=0\end{aligned}$$

Solving this equation gives us the value of t where intersection happens.

2.1.3 Blob

Based on the design of Google Blob emoji, I made these objects to have a rounded head that looks like half sphere, and a body with truncated-cone shape. The body and head are perfectly connected so that the whole object looks like an engineering helmet.

To find an intersection of a ray and a blob object, we first compute the intersection with a sphere that is centered at $(0,0,0)$ with

$$(O+t\times D)\cdot (O+t\times D)=1$$

Then we check if $O_y+t\times D_y$ is within the range where we want it to have the sphere shape. If not, we compute the intersection again with a truncated cone.

A truncated cone can be viewed as a cylinder with a slope added to its side. According to the discussion above, intersection with a cylinder requires

$$(O_x+t\times D_x)^2+(O_z+t\times D_z)^2=1$$

If we add a slope with angle $atan(5)$, then we need the intersection to satisfy

$$\sqrt{(x^2+z^2)}-1=\frac{y}{5}$$

$$\Rightarrow (D_x\times D_x+D_z\times D_z)\times t^2+(2\times (D_x\times O_x+D_z\times O_z)+\frac{D_y}{5})\times t+O_x\times O_x+O_z\times O_z-1+\frac{O_y}{5}=0$$

2.2 Texture mapping

Texture mapping is a method that allows us to render detailed images without constructing millions of polygons to shade complex object surfaces. To map a texture onto an object, we have to convert a 3D position in the scene into a 2D point in the texture image, and many factors need to be taken into consideration.

First, we want the image pattern in the texture to remain the same after mapping, for example, if it's a brick wall texture, we need the mapped object to look like a brick wall. This means that the texture has to be equally mapped onto object surface. For a plane, this might be as simple as removing the z value in its position, and normalize both x and y value to $[0,1]$. However, if surface is not flat, special handling is needed for the curved area. Such handling is not fixed for all objects with same shape, it is also affected by how the texture is like.

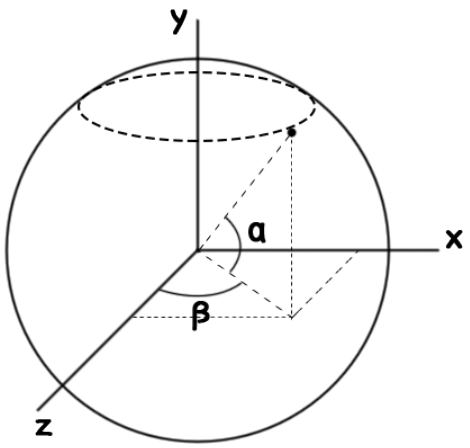
Secondly, how we compute (x,y) position is also related to the size of texture images. If the image is much larger than the rendered surface, we may map the surface to only a small portion of the whole texture. If we want the texture to repeat on the surface, we can map multiple different positions to the same (x,y) point, which requires the texture to be repetitive in some way (e.g., the left and right edge perfectly match).

In this project, I implemented 2D texture mapping for sphere, plane, cylinder and blob.

2.2.1 Sphere

One of the most common way to map texture onto a sphere is to use the inverse of Mercator Projection. If we consider the earth as a sphere, each location in the world can be represented in both cartesian coordinate system and geographic coordinate system (with longitude and latitude). Similarly, we can compute the longitude and latitude for each point on the sphere surface, and use that as the (x, y) coordinate in texture images.

(a) Convertion between Cartesian Coordinate and Geographic Coordinate



(b) Sphere with Wood Texture

Figure 1: Texture Mapping for Sphere

As in the image, α represents latitude, and β is longitude. If we define vector Y as $(0,1,0)$, and vector P as the vector from the origin to the point on surface, then we have the following equation:

$$\alpha = \frac{\pi}{2} - acos(Y \cdot P)$$

$$cos(\beta) = \frac{P_z}{cos(\alpha) \times |P|} + \frac{1 - \frac{P_x}{|P_x|}}{2} \times \pi$$

Now we have α in $[-\frac{\pi}{2}, \frac{\pi}{2}]$, β in $0, 2 \times \pi$, all we need is to normalize their range to $[0,1]$.

2.2.2 Cylinder

Mapping for cylinder can be considered as a conbimation of both plane and sphere. We can simply map y value to the range $[0,1]$, and for x and z, we can pick either of them to compute a longitude value the same way as we did for sphere.

2.2.3 Blob

The body of a blob can be considered as a transformed cylinder, and is thus easy to map textures on. However, because a blob has no detail on its body, in this specific case we actually can ignore the fact that its body is not a half sphere, and simply use the same mapping function for sphere on it.

2.3 Refraction

Refraction is a physical phenomenon where light travelling from on medium into another gets bent. The change in light direction can be computed based on the refractivity, or index of refraction, of the mediums.

Refracted colour can be computed by recursively following bent rays and calling the shading function on them until either we reach desired recursive depth or no object is hit by the ray. Once we get the refracted colour, we need to multiply it by some parameters. The most important parameter is affected by the refractive index. Not 100% of the light that hits a surface gets refracted, the rest is reflected on the surface instead of travelling through. The amount of light being reflected/refracted can be calculated using Fresnel Equation. According to Alexander I. Lvovsky's paper in 2013[1], the two Fresnel equations are as following:

$$R_p = \frac{\eta_2 \times cos\theta_1 - \eta_1 \times cos\theta_2}{\eta_2 \times cos\theta_1 + \eta_1 \times cos\theta_2}^2$$

$$R_t = \frac{\eta_1 \times cos\theta_2 - \eta_2 \times cos\theta_1}{\eta_1 \times cos\theta_2 + \eta_2 \times cos\theta_1}^2$$

where η is the index of refraction for mediums, θ_1 and θ_2 are the incoming angle and transmitted angle. Then, the refracted ratio is:

$$F_t = 1 - \frac{R_p + R_t}{2}$$

We can use this ratio to calculate how much reflected/refracted light should be perceived.

The other element that affects the refracted colour is transparency. Transparency is related to the change of IOR when wavelength is changing, which is out of the scope of this project. I defined a float variable in the PhongMaterial class that represents transparency or opacity and varies between 0 and 1. We multiply this amount onto the refracted colour as well.

2.4 Glossy reflection

Glossy reflection refers to the optical phenomenon where many objects cannot reflect as well as mirrors. The cause of this phenomenon is the imperfection of object surfaces. Perfect reflection can only occur on surface that is perfectly smooth, which is not quite common in the nature. Instead, the reflection we observe in our life usually looks hazy and blurred.

In ray tracing, it is hard to precisely create a rough surface and render reflection on top of it, instead, we can manually add in perturbation when we calculate reflection so as to simulate the roughness.

(a) Mirror Reflection (without Perturbation)

(b) Blurry Reflection (with Perturbation)

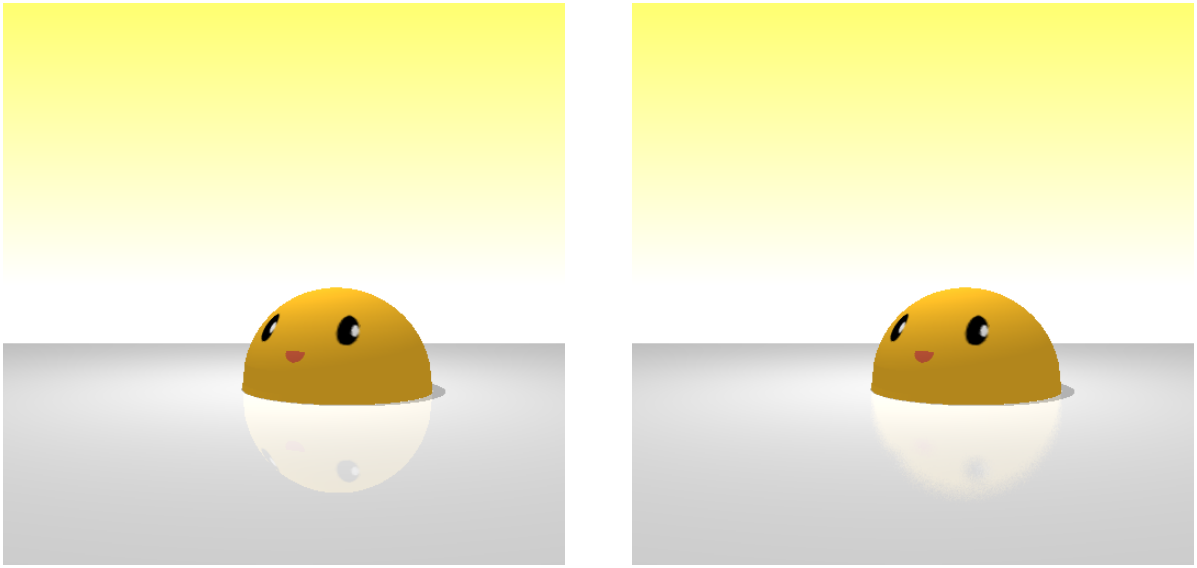


Figure 2: Glossy Reflection

In this project, I added a float number for each PhongMaterial to represent the glossiness of its surface. The computation of reflected colour is executed multiple times, each with a small random perturbation vector added into

the reflected ray direction. In this way, we mimic the situation where rough surface perturb the reflection angle, and produce blurry reflection on surfaces.

2.5 Anti-aliasing (Supersampling)

When we render an image where the pixel amount is not large enough, objects will have jagged edges and looks distorted. This is an effect called aliasing. The fact that our images are constructed by pixels means that we can never draw a continuous edge if it's not aligned with either x or y axis. However, we can reduce aliasing by applying supersampling technique.

Traditionally, we only shoot on ray for each pixel, which means that if we are drawing a black edge on white background, one pixel is either black or white, thus the final image looks jagged. By using supersampling, we shoot multiple rays for each pixel, and then take the average of all the results we get to retrieve the final value. In this way we allows a transition between black and white, even though each pixel still only has one colour, the whole image looks more natural and less jagged in our eyes.

In addition, I implemented adaptive anti-aliasing, which will compare the result from each sample, and if the difference in each sample is big enough, it will recursively execute anti-aliasing in that pixel again with smaller division, until it either reaches the maximum level or the result is close enough.

(a) A Line in Grid (b) without Anti-aliasing (c) with Anti-aliasing

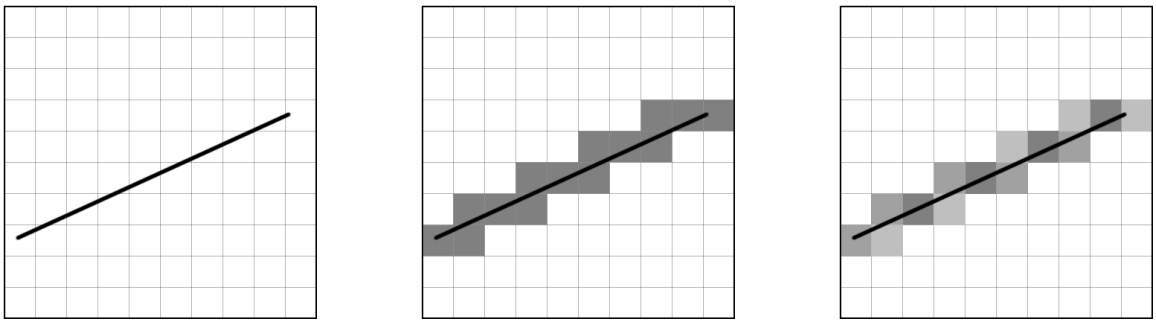


Figure 3: How Anti-aliasing Works

(a) without Anti-aliasing (b) with Anti-aliasing

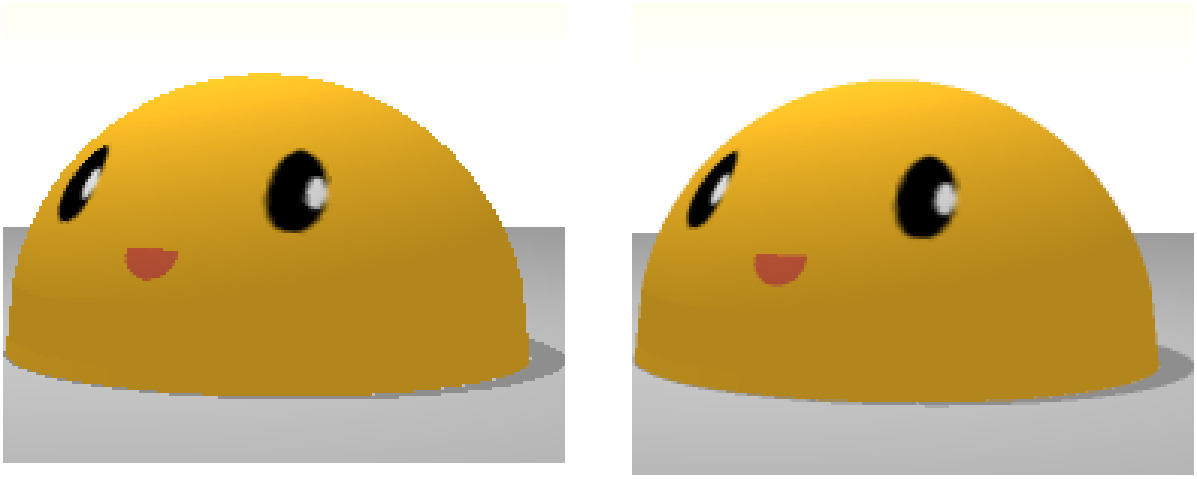


Figure 4: Anti-aliasing Used in Actual Scene

2.6 Soft shadows

In ray tracing, when a light source is blocked by another object, we do not add colour value to the surface being blocked by others, which at the end produces shadows on surface. The shadow is not soft shadow, meaning that it has a sharp edge , which is quite different from what we usually observe. The reaosn that shadows look sharp and ”hard” is that we are using point light source, so every area on every surface is either hit by light or blocked by others. Again, we can solve this problemby adding perturbation, the same as we produce glossy reflection.

This time the perturbance is added to the position of light source. In other words, we can imagine the light not as a point, but an area. Each time we test if a light can reach an area, we cast a ray randomly from that area to do the computation. At the end we take the average of all iterations, which gives us a soft gradient from shadow to lit-up area.

(a) Sharp Shadow (b) Soft Shadow with 50 Samples

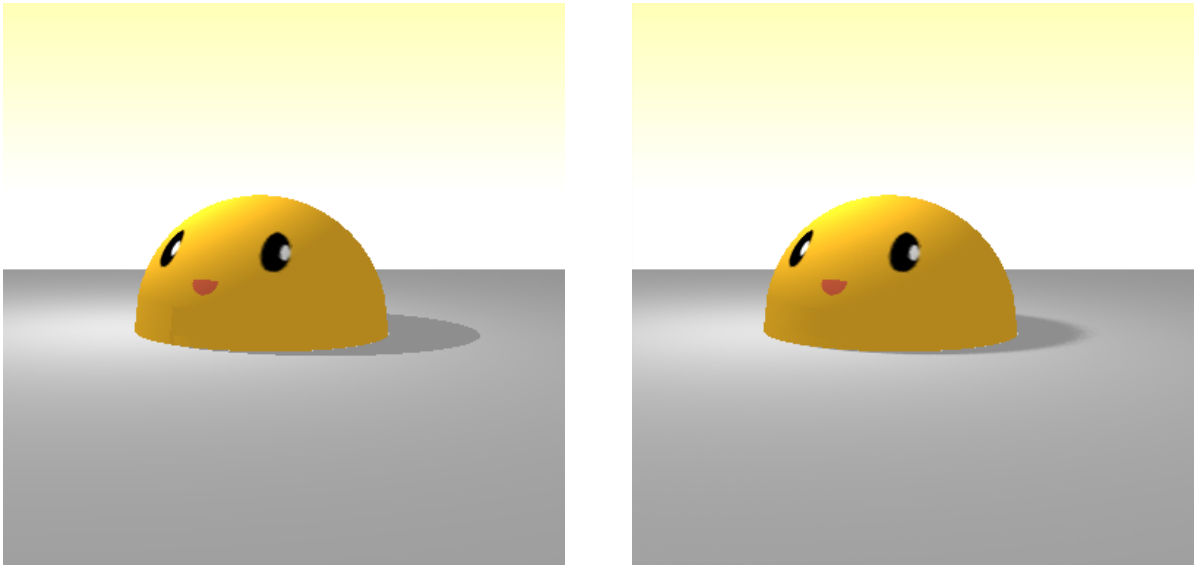


Figure 5: Soft Shadow

2.7 Motion blur

Human eyes can only process about 24 frames per second, this phenomenon is called the persistence of vision. Similarly, for cameras this rate is determined by its shutter speed. If objects are moving within the period of exposure, the photo will capture a blurry image of those objects, and we can tell from the photo that these objects are moving.

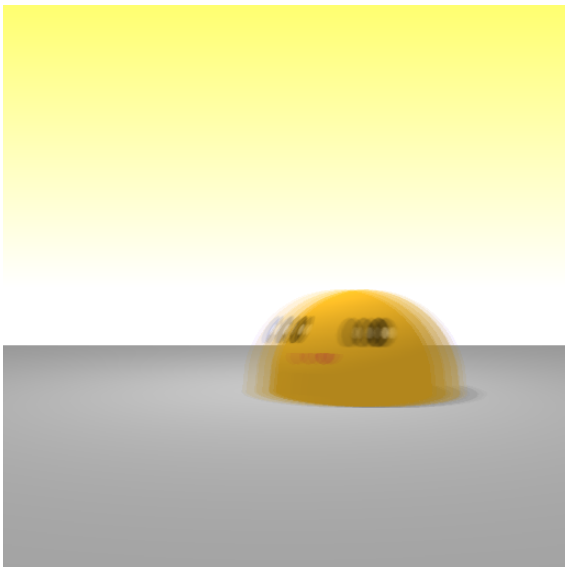
To produce such effect, we need the similar technique that we applied for rendering soft shadows or glossy reflection. We define a movement or deformation for an object, such as an equation to compute its location based on time, then for each sample we pick a random time frame, get the current position, trace rays to produce the image. After several passes, we average all the results to get the final scene.

2.8 Perlin noise

Perlin noise is developed by Ken Perlin to generate more natural noise in computer graphics. In his paper[2] in 2002 he described the algorithm as following:

Given a position defined by (x,y,z), we first pick a set of hash values for each of the corner in a unit cube. These hashes are picked based on the integer component of (x,y,z). Then we remove the integer component from coordinates to normalize the vector into a unit cube, and do a dot product on it with a gradient vector picked by the corresponding hash value. Values from dot product are passed into lerp function, at the end all these procedures produces one float number for each position. The number should be within $[0,1]$.

(a) Moving Blob



(b) Moving Ball

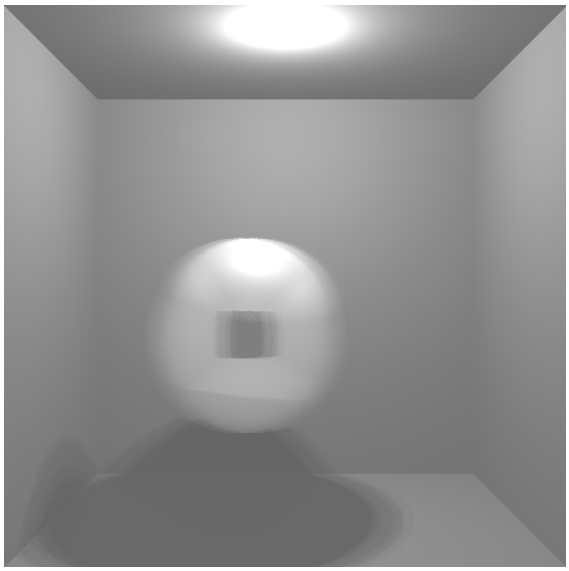


Figure 6: Motion Blur

Each computation of all points we need is called an octave. We can adjust the noise by setting different scale and weight value for each octave. Depending on the number of octaves we want, combining results from all octave computations produces a nice and natural noise map, which we can make use of in our own way.

In this project, I used perlin noise to colour the surface of a sphere to make it looks like the surface of moon.

Figure 7: Moon with Perlin Noise

3 Implementation

3.1 Algorithms

The most complicated algorithm implemented in this project is to produce perlin noise. This is implemented based on Perlin's paper in 2002, and also an online article written by Flafla2[3], which is also referencing Perlin's paper.

3.2 Modularity, Data Abstraction and Encapsulation

I created several classes to encapsulate different logic component and to group related information together.

1. Perlin: this class contains all logic needed to generate perlin noise, including the permutation table and multi-octave algorithm.
2. Primitives: each extra primitive I implemented lives in its own class. I also added a **getIntersection** function for each class, which takes in a ray and computes the intersection of it with a unit primitive.
3. Ray: this class is created to contain information of rays we shoot during rendering. It only contains an origin and a direction vector and is handy to use when ray is passed around to compute colours and intersections.
4. Texture: this is to store information about textures used in rendering. It's created by **gr.texture** command in Lua. It reads in bmp image information, and can return colour vector when given a 2D position.

I also added a **Helper.cpp** file to include some helper functions I built (such as float number comparison using EPSILON).

3.3 Global Constants and Configurability

The following constants are defined in **Project.cpp** and **Helper.cpp** to toggle on/off certain features and configure these features.

1. bool **REFLECTION_ENABLED**
2. bool **REFRACTION_ENABLED**
3. bool **ANTI_ALIASING_ENABLED**
4. bool **ADAPTIVE_ALIASING_ENABLED**
5. bool **SOFT_SHADOW_ENALBED**
6. bool **GLOSSY_REFLECTION_ENABLED**
7. bool **PERLIN_NOISE_ENABLED**
8. bool **MOTION_BLUR_ENABLED**
9. bool **GRID_ACCELERATION_ENABLED**: this is not implemented and has no effect.
10. double **PI=3.14159**: the ratio of a circle's circumference to its diameter.
11. double **LIGHT_SIZE= 50**: the maximum size of perturbation in rendering soft shadow.
12. double **EPSILON = 0.000001**: defined to replace zero to avoid error introduced by float point computation.
13. int **MAX_REFLECTION_LEVEL = 2**: the maximum recursive level in reflection and refraction;
14. int **ANTI_ALIASING_MAX_DIVISION = 5**: the maximum division level in anti-aliasing;
15. float **ANTI_ALIASING_MAX_TOLERANCE = 0.1**: the maximum difference in colour the program can tolerate in anti-aliasing. This variable and the one above are only used in adaptive anti-aliasing.

There is no interaction can be done in command line with the program. Toggling of additional features can be achieved by changing the variables mentioned above that are set at the top of **Project.cpp** file.

3.4 Lua Functions

Following functions are defined in **scene_lua.cpp**:

1. **gr_node_perlin_noise_cmd**: Set perlin noise on a node.
2. **gr_node_set_texture_cmd** : Assign a texture object to a node.
3. **gr_texture_cmd** : Create a texture object.
4. **gr_cylinder_cmd** : Create a cylinder primitive.
5. **gr_torus_cmd** : Create a torus primitive.
6. **gr_blob_cmd** : Create a blob primitive.

I also made changes to function **gr_material_cmd** so that it takes in more parameters such as IOR and opacity.

3.5 External Sources

There are two pieces of code that is from informations I found in paper and online resources. The first one is the computation of fresnel equation, it's based on Alexander's paper in 2013, and an online article in Scratchapixel 2.0's ray tracing series[4]. The second one is the computation of perlin noise, it's based on Perlin's 2002 paper and the online article *Understanding Perlin Noise*, the ones I mentioned above. Modifications are made to adapt these code pieces into the project.

Besides these, there is no code used from my work in previous terms. The whole directory structure is copied from Assignment 4, however, I rewrote most of the code in Assignment 4 to make the structure cleaner and make it easier to add in additional features.

4 Bibliography

1. Alexander I. Lvovsky (2013) Fresnel Equations. *Encyclopedia of Optical Engineering*, 1-6.
2. Perlin K (2002) Improving Noise. *ACM Transactions on Graphics, 21*(3), 681-682.
3. Flafla2 (2014) Understanding Perlin Noise. https://flafla2.github.io/2014/08/09/perlinnoise.html
4. Reflection, Refraction and Fresnel. *Introduction to Shading*, Scratchapixel 2.0. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel

* The latter two are online resources without peer review, they are cited because they helped me to better understand the original paper, and some of the code (mentioned in External Sources) I have in the project are modified based on their work.

5 Compilation

Compilation of the program remains the same, **premake4 gmake && make** should build the project in its root folder, and **./Project [lua-file]** can run the project.

6 Manual

To turn on a specific feature, we need to modify the corresponding constant defined in **Project.cpp** to true. This allows most features to be run with any given lua file. However, for motion blur, it's configured to only work on nodes with name "blob1".

There is no input needed for the program except the path to lua script. When executed, the program will print out the current progress of rendering, and keep updating the progress until it finishes execution.

7 Objectives

1. Extra primitive (torus, cylinder).
2. Texture mapping.
3. Refraction.
4. Glossy reflection.
5. Grid acceleration mechanism.
6. Anti-aliasing (Supersampling).
7. Soft shadows.
8. Motion blur.
9. Perlin noise.
10. Final scene.