Ray Tracing: A Waterfall

Name: Milena Bukal
Student ID: 20623852
User ID: msbukal

Purpose :

The purpose of this project is to achieve a realistic simulation of water in an atmospheric scene, while demonstrating a variety of combined ray tracing methods.

Introduction :

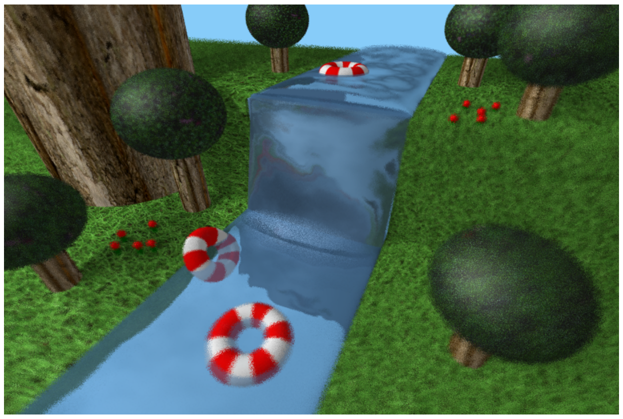
The final scene I aimed to achieve was of a waterfall in a forest. I also wanted to add other scenic elements that you would expect to see, such as grass and trees, to add more atmosphere to the scene. This is roughly inspired by a scene from a video game, but the scene I aimed to create was more vivid and aesthetic.

Old School Runescape: http://oldschool.runescape.com/

I implemented refraction and glossy reflection to simulate the way light acts when it hits the surface of the water. For the texture and waves on the surface of the water, I implemented bump mapping and used a Perlin noise function to generate the bump map. For the texture of other features, such as the grass and trees, I additionally implemented texture mapping. To create mist at the bottom of the waterfall, I added basic volumetric materials.

I also added an extra cylinder primitive, to be used for the base of the trees and the shape of the water, since the surface of water can be seen to be more curved. To add inner tubes, I also added a torus primitive. To improve the overall quality of my image, I additionally implemented anti-aliasing using super sampling, depth of field, and soft shadows.

Using these methods, I aimed to challenge myself in creating realistic water and achieving a high quality final scene. This was interesting to me as it involves many methods covered in lecture, therefore helping me learn their implementations more in depth, and resulted in one final demonstrative scene.



The Final Scene

A full summary of this progression in images, along with some small gifs, can be found at bit.ly/2YrQ20M. Texture sources can be found under the Bibliography.

Technical Outline :

Extra Primitives: Cylinder and Torus

I added two extra primitives to help construct my final scene: a cylinder and a torus.

Cylinder: The equation of a cylinder on the y axis of radius 1 centred at (0, 0, 0) is defined as:

$$x^2 + z^2 = 1$$

If o is our viewing point and d is the ray direction, and the intersection point can be calculated as $o + t \times d$, then we can solve for t by substituting the point in the equation above:

$$(o.x + t \times d.x)^2 + (o.z + t \times d.z)^2 = 1$$
$$t^2(d.x^2 + d.z^2) + t(2 \times o.x \times d.x + 2 \times o.z \times d.z) + (o.x^2 + o.z^2 - 1) = 0$$

This gives us two roots, $t0$ and $t1$. Consider $y0$ and $y1$, the y-values of the intersection with the cylinder for $t0$ and $t1$ respectively, where $y0 < y1$.

We need to restrict the cylinder to a fixed height of [-1, 1], If $y0$ and $y1$ are both less than -1, or both more than 1, then the ray has missed the cylinder. If $y0$ is within the bounds of [-1, 1], then the body has been intersected at $t0$.

Additionally, the intersection at the caps can be calculated. If $y0 < -1$ and $y1 > -1$, then the cap has been intersected at CAP $= -1$. If $y0 > 1$ and $y1 < 1$, then the cap has been intersected at CAP $= 1$. The value of t for the intersection of the cap can therefore be computed as:

$$t = t0 + (t1 - t0) \times (y0 - \text{CAP}) \div (y0 - y1)$$

[2]

Torus: A torus can be parameterized by two radiuses: R, the distance from the centre of the torus to the centre of the tube, and r, the radius of the tube itself. A torus can then be defined as:

$$(x^2 + y^z + z^2 + (R^2 - r^2))^2 = 4 \times R^2 \times (x^2 + z^2)$$

If o is our viewing point and d is the ray direction, and the intersection point is at $o + t \times d$, then we can solve for t by substituting the point in the above equation.

For convenience, define G, H, I, J, K, L as:

$$G = 4R^2(d.x^2 + d.y^2)$$
$$H = 8R^2(o.x \times d.x + o.y \times d.y)$$
$$I = 4R^2(o.x \times o.x + o.y \times o.y)$$
$$J = d.x^2 + d.y^2 + d.z^2$$
$$K = 2(o.x \times d.x + o.y \times d.y + o.z \times d.z)$$
$$L = o.x^2 + o.y^2 + o.z^2 + R^2 - r^2$$

Therefore, the above substitution can be expressed as:

$$J^2 \times t^4 + 2JK \times t^3 + (2JL + K^2 - G) \times t^2 + (2KL - H) \times t + (L^2 - I) = 0$$

Using a quartic root solver, the minimal positive t is the intersection point of the ray with the torus. [3]

Anti-Aliasing (super sampling)

To improve the overall quality of my image, and to reduce jagginess, I added anti-aliasing using super sampling. This method involves shooting multiple rays for each pixel in the image and averaging them. To do this, points close to the original pixel are created and then used to shoot rays. The resulting colours of these rays are averaged. [1]

If regular sampling is used, with defined n^2 pixels around a centre pixel (i, j), more artifacts may be introduced due to regular patterns. [11] To reduce this noise, I will jitter these points to generate random variation on them.

I compute a jittered pixel (ji, jj) as:

$$\mathrm{ji} = \mathrm{i} + (p + \mathrm{RAND}[0,1] - 0.5)/n$$

$$\mathrm{jj} = \mathrm{j} + (q + \mathrm{RAND}[0,1] - 0.5)/n$$

for $p, q = 0$ to $n - 1$. The result colour at pixel (i, j) is then the average of all the jittered pixels.

Depth of Field

A human eye is not a single point opening but instead an area that can only focus on a single depth at a time. Things out of this frame of focus are blurry. This depth of field can be simulated in ray tracing, by representing the origin of the rays, the "eye", as a lens. This should be done using a disk, however I will use a square lens as a sufficient approximation. [10]

Therefore, we define a square around the eye to mimic the lens, and pick random points on this lens to use as the new origin for rays. The position of the eye will be perturbed using points selected randomly, to get a random distribution of rays.

Soft Shadows

To make the scene a more natural, soft light, I additionally implemented soft shadows. In real life, light does not come from one fixed point, but instead an area such as the sun or a lamp. This causes softer shadows, where the intensity of the shadow varies. [10]

To simulate this using ray tracing, for a light at point l with orthonormal basis (u, v), I model the perturbed light point l' as:

$$l' = l + \mathrm{RAND[0,1]}u + \mathrm{RAND[0,1]}v$$

By selecting n random shadow rays, and then averaging the results, these rays will mimic light coming from an area instead of one fixed point, generating soft shadows.

Glossy Reflection

For simulating light reflecting on water, I implemented glossy reflection as an add-on to my prior implementation of mirror reflection. To implement glossy reflection, the reflected ray r is perturbed to generate a glossy reflected ray r' by randomly sampling a square area around r of size a. [10]

I compute the glossy reflected ray r' as:

$$r' = r + (a \times (\mathrm{RAND}[0,1] - 0.5))u + (a \times (\mathrm{RAND}[0,1] - 0.5))v$$

where (u, v) make an orthonormal basis for the reflected ray r.

Refraction

For simulating the way light travels from air in to water, I implemented refraction. When light travels from one medium with refractive index i_1 and then through a refractive medium with a different refractive index i_2, it bends but continues traveling through the medium.

By Snell's law, we know that for incoming angle θ and outgoing angle ϕ:

$$i_1 sin\theta = i_2 sin\phi$$

Given an incoming vector d from a medium with a refractive index i_1, which then intersects an object with surface normal n and refractive index i_2, I can compute the outgoing refracted ray r as:

$$r = \frac{i_1(d + ncos\theta)}{i_2} - ncos\theta$$

Which is equivalent to:

$$r = \frac{i_1(d - n(n \cdot d))}{i_2} - \left(n\sqrt{1 - \frac{i_1^2(1-(d \cdot n)^2)}{i_2^2}} \right)$$

[9]

Combining Reflection and Refraction

As a material can be both reflective and refractive, I use Schlick's approximation:

$$R(\theta) = R_0 + (1 - R_0)(1 - cos(\theta))^5 \text{ where } R_0 = \left(\frac{i_2 - i_1}{i_2 + i_1}\right)^2$$

to determine how much of the ray is reflected and how much is refracted:

$$\text{colour} = R(\theta) \times \text{reflected} + (1 - R(\theta)) \times \text{refracted}$$

[7]

Texture Mapping

To add textures to my scene, such as grass, leaves, and bark, I added texture mapping. A texture map is a provided 2D image with texture coordinates (u,v). These texture coordinates can then be looked up, and used to colour a given point.

For a sphere, the (u, v) value has to be computed to be scaled around the circular shape of sphere. If a sphere is intersected at point (x, y, z), then we can compute spherical angles:

$$\theta = arcsin(y) \text{ and } \phi = arctan(z, x)$$

and then compute (u,v) with them as:

$$u = 0.5 - \phi/(2\pi) \text{ and } v = 0.5 - \theta/\pi$$

[4, pp. 29-31]

By scaling (u, v) coordinates in similar ways, this can be extended to further primitives such as cubes, toruses, and cylinders.

Bump Mapping

Bump mapping was used to form rippling on the surface of running water. Bump mapping pertrubs surface normals to form the appearance of "wrinkles", according to a provided perturbation function $B(u, v)$. This function can be defined using a bump map file, such as a texture map, or can alternatively be supplied through a Perlin noise function, which is what I used for simulating water.

Given a point P with normal n, the new normal n' can be defined as:

$$n' = n + Bv(n \times Pu) + Bu(n \times Pv)$$

where Bu, Bv are partial derivatives of the bump function B, and Pu, Pv are the partial derivatives of p, with respect to u, v. [5]

Perlin Noise

To create a rippling function for water to use with bump mapping, I used Perlin noise. Perlin noise is used to generate "noise" in an image that appears random. This can be used to simulate the "randomness" of water and its waves.

The Perlin noise function specifically is a method that generates "noise" for a given point. It uses a permutation function to interpolate a real point based off the gradient of surrounding integer points. This results in a given "height" or perturbation.

Perlin noise can be used to simulate the frequency needed for waves by looping over octaves:

$$\sum_i \frac{\text{Noise}(\text{point} \times 2^i)}{2^i}$$

[6]

Bonus: Mist - Volumetric Material

At the bottom of a waterfall, you see mist from water particles in the air. Simulating each particle is unpractical, so instead we visualize the "mist region" as a volume with a probability for the distance that a ray travels in to it before it scatters. The probability that the ray scatters in a small distance dL can be defined as:

$$\text{probability} = \text{density} \times dL$$

and therefore, the distance it travels can be modeled as:

$$\text{distance} = \left(\frac{-1}{\text{density}}\right) \times \text{RAND}[0, 1]$$

[4, pp. 45-48]

Implementation :

Note: I never tested or implemented any of these features with non-hierarchical primitives, as I only used them originally to ensure ray intersection was working before I added hierarchy.

Extra Primitives: Cylinder and Torus

To add extra Primitives, I extended the `Primitive` class with two new subclasses: Cylinder and Torus, with the corresponding `Primitive::intersect(...)` method.

The cylinder does not have any additional fields. I have already defined it to be centred at (0, 0, 0), with radius 1 and height bounded by [-1, 1]. To create other cylinders, it can be scaled in the x, y, z directions hierarchically. However, for the torus, scaling would not be able to change proportion of it's two radiuses R and r. Therefore, the Torus class additionally takes the required parameters R and r.

To create cylinders and toruses, I added the new lua commands `gr.cylinder(name)` and `gr.torus(name, R, r)`, which create Geometry Nodes with `name` of the corresponding primitive type.

Ray Averaging Methods

There are three main ray averaging methods I added to improve the quality of the image: anti-aliasing, glossy reflection, depth of field, and soft shadows.

All of these methods use more than one ray for one result, and are computationally intensive. Therefore, to be able to render simple scenes quickly and to dynamically change the number of rays without having to re-compile, I extended the lua command:

```
gr.render(ROOT_NODE, OUTPUT_FILE, WIDTH, HEIGHT,
          EYE, VIEW, UP, FOV, AMBIENT, {LIGHTS})
```

to have 5 additional fields, which represent the number of rays used for each corresponding feature, along with the edge length of the lens for depth of field:

```
gr.render(ROOT_NODE, OUTPUT_FILE, WIDTH, HEIGHT,
          EYE, VIEW, UP, FOV,
          ANTIALIASING_RAYS, REFLECTION_RAYS, SHADOWS_RAYS, DEPTH_RAYS, DEPTH_LENS,
          AMBIENT, {LIGHTS})
```

To generate the ray traced image, the main method `A5::render()` calls the method `A5::getColour(...)` for each pixel (i, j) in the image.

Depth of Field

In `A5::getColour(...)`, if there is only one ray for depth of field, then depth of field is not enabled. If there is $n > 1$ rays, then n random rays are created within the dimension of the lens. The colour for each of these rays is found, and then averaged, to simulated Depth Field.

Within the method `A5::getColour(...)`, each ray returns a colour by calling the method `A5::getAntiAliasingColour(...)`.

An issue I encountered was that all objects in the scene were blurry with depth of field enabled. I realized this was due to me normalizing the distance between the eye point and the view point when computing the pixel (i, j) in world coordinates. Instead of changing the matrix I use for this from the previous assignment, I instead re-scaled the computed point pWorld:

$$\text{pWorld} = \text{eye} + (\text{pWorld} - \text{eye}) \times \text{length}(\text{view} - \text{eye})$$

Anti-Aliasing (super sampling)

In `A5::getAntiAliasingColour(...)`, if there is only one anti-aliasing ray, then anti-aliasing is not enabled. If there is $n > 1$ rays, then n^2 rays are generated in an $n \times n$ grid around the original pixel. They are additionally jittered slightly to avoid regular error.

Within the method `A5::getColour(...)`, each ray returns a colour by calling the method `A5::intersect(...)`, which is responsible for finding intersections for the provided ray with all objects in the scene, and then returning the found colour.

An issue I encountered was that my antialiased image was slightly shifted compared to the original. I noticed this was because I was computing my jittered points as:

$$\mathrm{ji} = \mathrm{i} + (p + \mathrm{RAND}[0,1])/n \quad \text{and} \quad \mathrm{jj} = \mathrm{j} + (q + \mathrm{RAND}[0,1])/n$$

and therefore it was shifting them only up and to the right. Therefore, I added an extra -0.5 to them, so that sample points surrounded the original point:

$$\mathrm{ji} = \mathrm{i} + (p + \mathrm{RAND}[0,1] - 0.5)/n \quad \text{and} \quad \mathrm{jj} = \mathrm{j} + (q + \mathrm{RAND}[0,1] - 0.5)/n$$

This allowed me to do exact comparisons when anti-aliasing was on and off.

Soft Shadows

Within `A5::intersect(...)`, I call `A5::directLight(...)` which computes the illumination on a point of intersection. If the object intersects another one before reaching a light, it is shaded. Otherwise, illumination calculations occur.

If there is only one shadow ray, then soft shadows is not enabled. If there is $n > 1$ shadow rays, then n random rays are generated near the point of the light, and used to compute shaded areas in the scene. Regardless, each shadow ray calls `A5::computeShadowRay(...)` to return the illumination on the provided point.

Glossy Reflection and Refraction

For glossy reflection and refraction, I extended the original `PhongMaterial` class to have additional fields for the reflection coefficient (how reflective a material is), the reflection area (the area used for the glossiness of the reflection, where 0 represents mirror reflection), and the refractive index of the material (where -1 means non-refractive).

I correspondingly extended the default lua command:

```
    gr.material(KD, KS, shininess)
```

to create a new command for a reflective and refractive material:

```
    gr.ref_material(KD, KS, shininess,
                    REFLECTION_COEFF, REFLECTION_AREA, REFRACTIVE_INDEX)
```

In the method `A5::intersect(...)`, the intersection point with the ray is found and shaded using `A5::directLight(...)`. If the total number of hits has not been exceeded, the number is incremented and the method `A5::intersectReflectionRefraction(...)` is called. In this method, if the object has a reflection coefficient greater than 0, then `A5::intersectReflection(...)` is called. If the object has a refractive index not equal to -1, `A5::intersectRefraction(...)` is called.

In `A5::intersectReflection(...)`, the reflected ray is computed and then recursively calls `A5::intersect(...)` to compute the colour from the reflected ray. If the object has a reflection area > 0, and therefore is glossy, multiple glossy reflected rays are cast in the same way as one reflected ray, and then averaged.

In `A5::intersectRefraction(...)`, the refracted ray is computed. If total internal reflection is happening, where the value in the square root would be negative, then the ray is not used. Otherwise, the refracted ray recursively calls `A5::intersect(...)` to compute the colour from shooting the refracted ray.

An assumption I made is that the original eye to pixel ray always begins with a refractive index of 1.0. This means that the eye can not be within a refractive material, or the refraction will not be correctly computed. This does not affect the scene I desired, where the waterfall is just a landscape, but could cause issues in other scenes.

In `A5::intersectReflectionRefraction(...)`, if only one of `A5::intersectReflection(...)` or `A5::intersectRefraction(...)` return a result, then that is the result that is added to the current computed colour. However, if both return a result, then the colour from the reflected and refracted ray is combined using Schlick's approximation as previously defined.

Texture Mapping

For texture mapping, I used the provided `lodepng` library to read in (r, g, b) values from a provided image.

I then created a new `Texture` class, which I added as a field of the `PhongMaterial` class. I added a bool `m_texture`, to know when texture mapping was available on an object, and a pointer a the `Texture` class instance.

Now, in the method `A5::intersect(...)`, I check if the material is texture mapped, if so call `Texture::texturevalue(u, v)`. The method `Texture::texturevalue(u, v)` computes the colour at the given (u, v) position by interpolating nearby points in the texture map. The value it returns can then be used as the KD value (the diffuse value) for the intersected object.

I compute the (u, v) coordinates in `Primitive::Intersect(...)`, at the same time that I compute ray intersections with objects in the scene. I extended my `Intersection` class to store these (u, v) values. This needed was because the values needed to be computed before transformations are applied.

For texture mapping, I added a new lua command:

```
gr.texture_material(FILE_PATH, KS, shininess,
                    REFLECTION_COEFF, REFLECTION_AREA, REFRACTIVE_INDEX)
```

which extends the previous `gr.refl_material`, but instead of taking a KD value, asks for the relative path to a texture map. As I used the `lodepng` library, only pngs can be used for texture mapping.

Once I had texture mapping working for spheres, I decided to additionally try and texture map my three other primitives.

For cylinders, I discovered that the same theta and phi angles as spheres works correctly, and the (u, v) coordinates can be calculated in a very similar way. For toruses, I used a source I found online [8]. For cubes, I realized that one of the (x, y, z) coordinates always equaled 0 or 1. Therefore, the (u, v) coordinates were determined according to scaling the other two coordinates.

Bump Mapping

For bump mapping, I extended the `Texture` class to have an additional `Texture::bumpvalue(...)` method. I added a field `m_bump` to `PhongMaterial`, so that in `A5::intersect(...)`, I can check if the material should be bumped map, and if it is then I apply the perturbation to the normal.

In `Texture::bumpvalue(u, v, ...)`, I use the same (u, v) coordinates as in texture mapping, and get my perturbation value by calling `Texture::bumpperturb(u, v)`. In this method, there are two options for the source of the perturbation for the normal. If `use_perlin` is enabled, then a value is computed using Perlin noise. Otherwise, the (r, g, b) value from a provided texture map image is averaged to be the perturbation.

To compute Bu and Bv, I found a small sample for the rate of change:

$$Bu = \text{bumpperturb}(u+1, v) - \text{bumpperturb}(u, v)$$

$$Bv = \text{bumpperturb}(u, v+1) - \text{bumpperturb}(u, v)$$

I computed Pu as the tangent point on a sphere:

$$Pu = (-sin(\phi), cos(\phi), 0)$$

and Pv as $Pv = n \times Pu$ for surface normal n.

As I had previously computed the phi angles of the sphere, cylinder, and torus, I was able to re-use these, store them in my `Intersect` class, and compute Pu and Pv for all three primitives in the same way described above. As my cube always has one (x, y, z) coordinate equal to 0 or 1, I used this point for the "phi" angle for bump mapping. This is more of a hack, as a cube is not circular, and therefore it does not work on all rotational angles of the cube.

For bump mapping, I added two new lua commands. The first is:

```
gr.bump_material(FILE_PATH, KD, KS, shininess,
                    REFLECTION_COEFF, REFLECTION_AREA, REFRACTIVE_INDEX, POS)
```

This command creates a bump mapped material which uses a provided texture map image for perturbations in the relative FILE_PATH. If POS = 1, then KD values for the object are retrieved using texture mapping, from the texture map image. If POS = 2, then then the KD value provided is used.

The second lua command is:

```
gr.perlin_bump_material(KD, KS, shininess,
                            REFLECTION_COEFF, REFLECTION_AREA, REFRACTIVE_INDEX,
                            Z_HEIGHT)
```

This command creates a bump mapped material which uses a Perlin noise function for perturbation of the normal with Z_HEIGHT, described in more detail in the next section.

Perlin Noise

To implement Perlin noise, I added a `Perlin` class within the file `Texture.hpp` that takes a `z_height` and has a 3D `Perlin::Noise(vec3)` method. I decided through testing that 5 octaves generated a good resemblance of water when using the Perlin noise to bump map.

As my Perlin noise function is used for bump mapping, which has a 2D texture coordinate (u, v), I initially only implemented a 2D `Perlin::Noise(vec2)` method. However, as the perturbation map `P` I use in my `Perlin` class is static, I could not easily change the result from my `Perlin::Noise(vec2)` method for the same input.

I realized that if I extended it to `Perlin::Noise(vec3)`, I could specify the z coordinate, the "`z_height`", as user input. Now, I call the method as `Perlin::Noise(vec3(u, v, z_height))`. This allows me to change the result by a small amount, and therefore smoothly change the value that I perturb the normal by in bump mapping.

As a result of this, I can create a rippling animation of the bump map on a surface to create the appearance of waves. I was additionally able to demonstrate that the Perlin noise function was being used for bump mapping, instead of just a bump map file, by animating it this way.

There is no direct command to create an instance of the `Perlin` class, but it can be specified using the previously defined `gr.perlin_bump_material(..., Z_HEIGHT)` command. Using this command, I can define the z value as `Z_HEIGHT` that is used to create a point for my `Perlin::Noise(vec3)` computations.

Bonus: Mist - Volumetric Material

To the class `PhongMaterial`, I added the field `m_density` for volumetric materials. In my `GeometryNode::intersect(...)` method, I check if value of `m_density` is non-zero. If it is, I randomly increase the t intersection result as specified previously, moving the point of intersection in to the intersected object.

However, an issue I encountered was that all of the points within the object were shaded, as they were “blocked” from reaching light sources. This was caused due to the shadow ray intersecting the original object itself on the way out, as the point of intersection had been moved in to the object. Therefore, I disabled shadows for volumetric materials, as it makes sense for mist not to be shadowed.

I added the corresponding lua command:

```
gr.volumetric_material(KD, KS, shininess, DENSITY)
```

where `DENSITY` must be a value in (0, 1].

Bonus: Plane

To make scenes to demonstrate my features simpler to create, I also added a Plane primitive with a corresponding `gr.plane(name)` lua command. The default plane is defined at point (0, 0, 0) with normal (0, 1, 0), and I computed an intersection as:

$$t = (n \cdot (q - \text{ray origin}))/(n \cdot \text{ray direction})$$

Bonus: Animation

To be able to generate each frame in my short animations non-manually, I needed to be able to pass a FRAME number as command line input to my C++ executable, which would then be used in my lua files. I declared the integer FRAME within `A5.hpp` with a default value of 0, and assigned it a value in `Main.cpp`, if there was provided command line input. I then added the lua command `gr.frame_cmd()`, which returns the value of FRAME within a lua file. Therefore, in my lua model files, I can access the current frame.

For example, to animate my Perlin noise on a sphere, I made the `Z_HEIGHT` of my material `gr.perlin_bump_material(...)` equal to FRAME / 100. I can then call my program for the value FRAME = 1 to 100, generating frames of smoothly changing Perlin noise. This can be seen in the files `perlin-sphere.lua` and `sphere.sh`.

Compilation and Manual :

To compile the program, the following commands need to be run in the cs488/ and cs488/A5 directory:

```
$ premake4 gmake
$ make
```

Then to run the program, within the directory cs488/A5/Assets, execute:

```
$ ../A5 FILENAME FRAME
```

where FILENAME is a lua file specifying a scene, and FRAME is an optional parameter for the current frame number.

The code was tested on lab computer GL13.

The lua file must be specified using valid lua commands from `scene_lua.cpp`. The new commands have been explained in the Implementation section.

The image is rendered using the following lua command:

```
gr.render(ROOT_NODE, OUTPUT_FILE, WIDTH, HEIGHT,
          EYE, VIEW, UP, FOV,
          ANTIALIASING_RAYS, REFLECTION_RAYS, SHADOWS_RAYS, DEPTH_RAYS, DEPTH_LENS,
          AMBIENT, {LIGHTS})
```

where all nodes in the scene are children of the ROOT_NODE.

The image will be created with dimensions WIDTH by HEIGHT in the file name OUTPUT_FILE. You can additionally specify the EYE, VIEW, and UP points in 3D, along with a value for FOV. The ambient light can be set as the value AMBIENT, and the scene contains the provided list of LIGHTS.

Additionally, you must specify the number of rays for anti-aliasing, glossy reflection, soft shadows, and depth of field, along with the size of the depth of field lens. The number of rays for each feature has to be strictly greater than 0. If the number equal to 1, the feature is disabled. If the number is greater than 1, the feature is enabled with that number of rays.

To track progress, every 50 lines drawn the current y position is printed, where $y = 0$ to $(\text{HEIGHT} - 1)$.

Bibliography :

[1] Crow, Franklin C. The Aliasing Problem in Computer-Generated Shaded Images. Communications of the ACM, vol. 20, no. 11, 1977, pp. 799805., doi:10.1145/359863.359869.

[2] Penfold, Dom. “Cylinder Intersection.” woo4.me, 29 Jan. 2014, woo4.me/wootracer/cylinder-intersection/.

[3] Cross, Don. “The Torus Class.” Fundamentals of Ray Tracing, 2013, pp. 150-153, cosinekitty.com/raytrace/raytrace_ebook.pdf.

[4] Shirley, Peter. Ray Tracing: The Next Week. 1.42 ed., 2018.

[5] Blinn, James F. “Simulation of Wrinkled Surfaces.” Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH 78, vol. 12, no. 3, 1978, pp. 286-292., doi:10.1145/800248.507101.

[6] Perlin, Ken. “An Image Synthesizer.” ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, 1985, pp. 287-296., doi:10.1145/325165.325247.

[7] Schlick, Christophe. An Inexpensive BRDF Model for Physically-Based Rendering. Computer Graphics Forum, vol. 13, no. 3, 1994, pp. 233246., doi:10.1111/1467-8659.1330233.

[8] Skiba, Wlodzimierz ABX. UV Mapping for Torus Object. abx.art.pl/pov/patches/uvtorus.php.

[9] Kay, Douglas Scott, and Donald Greenberg. Transparency for Computer Synthesized Images. Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH 79, 1979, doi:10.1145/800249.807438.

[10] Cook, Robert L., et al. Distributed Ray Tracing. ACM SIGGRAPH Computer Graphics, vol. 18, no. 3, 1984, pp. 137145., doi:10.1145/964965.808590.

[11] Chiu, Kenneth, et al. Multi-Jittered Sampling. Graphics Gems, 1994, pp. 370374., doi:10.1016/b978-0-12-336156-1.50045-8.

Texture Sources :

Grass: http://www.myfreetextures.com/wp-content/uploads/2015/01/grass-free-texture.jpg

Bark: https://freestocktextures.com/texture/bark-nature-wood,35.html

Leaves: https://freestocktextures.com/texture/green-leaves-background,998.html

Red and White Stripes: https://marketplace.secondlife.com/p/Red-and-White-Stripes-Texture/3651210?id=3651210slug=Red-and-White-Stripes-Texture

Donut: https://www.clipart.email/download/1808316.html

Brick: https://freestocktextures.com/texture/brick-wall-renovated,760.html

Earth: http://www.djcline.com/wp-content/uploads/2017/03/WorldmapMercator.jpg

Dots: http://math.hws.edu/graphicsbook/source/webgl/textures/dimples-height-map.png

Objectives: **Full UserID: msbukal** **Student ID: 20623852**

___ 1: Extra Primitives: Cylinder and Torus.

___ 2: Antialiasing (super sampling).

___ 3: Depth of Field.

___ 4: Glossy Reflection.

___ 5: Refraction.

___ 6: Soft Shadows.

___ 7: Texture mapping.

___ 8: Bump Mapping.

___ 9: Perlin Noise.

___ 10: Final Scene.

A4 extra objective: Mirror Reflection