CS488 Final Project Report
Title: Extended Ray Tracer
Name: Tengyu Cai
Student ID: 20494373
User ID: t7cai

Final Project:

Purpose :

The purpose of this project is to extend the raytracer created as a part of Assignment 4. A custom scene will be modeled as lua files to be passed in the ray tracer and the output high-realism images.
I finished all the objectives except that I changed the final scene described in the project proposal because the proposed final scene was much harder than I expected. Also, since I implemented full photon mapping, it would be better to render a scene in a closed space rather than in a open space as proposed.

Topic :

- Photon Mapping.
- Advanced Raytracing and Light Interaction Techniques.
- Using Fine Arts concepts to create a Final Scene.

Statement :

I model a cornell box with some clear water filling up nearly half of the box. Then I also have several transparent and reflective objects (incluing a reflective sphere, a transparent glass ball and a medal ring) lying under water. I made the floor and the wall of the cornell box with bumpy texture to make it looks more realistic and beautiful.

Below is a brief summary of what I implemented for the extended ray tracer to achieve this final scene. More implementation details can be found in implementation section.

1. **Extra Primitives**
   The basic ray tracer in Assignment 4 supports spheres and cubes. I supported cylinder as my extra primitive.

2. **Constructive Solid Geometry**
   In order to model the medal ring, I implemented csg and use the difference command with two different size cylinder to create a ring.

3. **Texture Mapping**
   I implemented texture mapping to map some tiles and rock images to the wall and the bottom of the cornell box. So that the final scene will not looks like the classic cornell box with just pure colours.

4. **Bump Mapping**
   I implemented bump mapping and applied it to the texture surfaces so that the texture surfaces on the final scene will not looks like a flat image stick on them but have bumpy surfaces that showing surface details.

5. **Photon Mapping**
   Photon mapping is the main topic of my extended ray tracer. I didn't use the normal ray tracing or combine it with the photon mapping but implemented a full photon mapping only. I intended to use photon mapping to achieve a better visual effect than the normal ray tracer and spend lots of time on it.

6. **Caustics**
   Caustics occur when light is focused at a certain point: light goes through a number of specular bounces, only to reach a diffuse surface. Caustics is one of the main reason that I implemented photon mapping, as this effect is beautiful but not able to achieve using ray tracing. I demonstrate this effect on my final scene through the refraction of the glass ball and the water surface.

7. **Diffuse Interreflection**
   Diffuse interreflections come from light that bounces around the scene, "grabbing" color at each step. This is another reason that I implemented photon mapping so that I can have a global illumination effect that are not able to achieve using normal ray tracing.

8. **Area Light Source**
   I added a area light source after I realized that a point light source can not give a very good result with photon mapping. The image rendered with point light source had lots of noise even with a large number of photon emitted. But with area light sources, this problem is solved.

9. **Soft Shadow**
   I implemented soft shadow to make the scene looks more realistic. Soft shadows are better than sharp shadows because there are seldom point light source in our real life.

10. **Refraction**
    I added refraction properties to the material so that I can use transparent objects to show different patterns of caustics. Also, different materials with have different refractive index which will make the scene more interesting.

11. **Perlin Noise**
    After I added a water surface, I noticed that if I use too many triangles to simulate the wave it will take a long time to render the image. However if I don't use enough triangles, the surface will look odd with lots of visible triangles. So I tried use fewer triangles to reduce the rendering, but at the same time, I added some perlin noise to each triangle to make each face smoother.

12. **Adaptive Anti-aliasing**
    Anti-aliasing can help with reducing the "jaggies" in the scene which makes the final image looks more smooth and elegant.

13. **Multi-threading**
    Since the rendering time for a high resolution image is very long, in order to speed up the rendering process, I supported the ray tracer to use multiple threads on computers with multi-core processor.

Additional LUA commands :

- `gr.cylinder(<name>)`: Create a cylinder with centre (0,0,0), radius 1 and height 1.

- `gr.union(<name>, n1, n2)`: Create a CSG union node with node n1 and node n2.

- `gr.intersection(<name>, n1, n2)`: Create a CSG intersection node with node n1 and node n2.

- `gr.difference(<name>, n1, n2)`: Create a CSG difference node with node n1 and node n2 (n1 - n2).

- `gr.light({x,y,z}, {r,g,b}, {c0,c1,c2}, p, num_photons)`: Create a point light source located at (x, y, z) with intensity (r, g, b), quadratic attenuation parameters c0, c1 and c2 and power p. num_photons indicates the total number of photons that will be emitted from this light source.

- `gr.material({dr, dg, db}, {sr, sg, sb}, p, t, r, s, <texture>, <bump>)`: Return a material with diffuse reflection coefficients dr, dg, db, specular reflection coefficients sr, sg, sb, and Phong coefficient p. The newly added fields are transparency t and refractive index r of the material. The Perlin Noise scale s can add some Perlin Noise to the surface of this material. Also, by providing the path to the texture image and/or bump image, the material can be render with a texture and/or bumpy surface.

Implementation :

1. **Extra Primitives**
   Relevant code is located in `Primitive.hpp`, `primitive.cpp` and `scene_lua.cpp`.

   I added a new command `gr.cylinder(<name>)` to support Cylinder. This command will created a Cylinder with its center point located at the origin. To implement this extra primitive, I created a new Cylinder class in `Primitives.hpp` and `Primitives.cpp`.

   To test intersection of ray with the cylinder side, I use the equation of an elliptic introduced in the course note: $\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$. I also tested intersection of ray and circle surfaces for the top and bottom faces of the Cylinder. Finally, I used the nearest intersection point as the intersection of the ray and the new primitive.

   Note that because I have implemented hierarchical transformation, the intersection tests will be really simple because all cylinders, when testing intersection, will be at the origin.

2. **Constructive Solid Geometry**
   Relevant code is located in `CSGNode.hpp`, `CSGNode.cpp` and `scene_lua.cpp`.

   I extended the LUA command with three new commands in `scene_lua.cpp`: `gr.union(<name>, <node1>, <node2>)`, `gr.intersection(<name>, <node1>, <node2>)` and `gr.difference(<name>, <node1>, <node2>)` to support *union*, *intersection* and *difference* respectively.

   I created a new virtual base class `CSGNode` that inherited from `GeometryNode` with two additional Geometry Node properties. Its subclasses (UnionNode, IntersectionNode and DifferenceNode) all have their own implementation of `intersect(eye, ray)` that return the intersection point of the CSG.

For union node, I use the nearer point return from two geometry nodes in the case that either one of them is hit.

For intersection node, I use the nearer point return from two geometry nodes in the case that both of them are hit.

For difference node, in the case that both two geometry nodes are hit, I perform two new intersection tests will eye positions set to the previous intersection points. In this way, I can get a range of the intersection and perform a boolean operation these line segments.

3. **Texture Mapping**
   Relevant code is located in `Texture.hpp`, `Texture.cpp`, `PhoneMaterial.hpp`, `PhoneMaterial.cpp`, `Primitives.hpp`, `Primitives.cpp`, and `scene_lua.cpp`.

   I created a new Texture class that reads and stores the information of the texture image so that we can easily retrieve the colour of the image given the UV coordinates. For each material, if initialized with a path to a texture image, will have an additional `Texture *m_texture` field that points to the Texture class. Then in the `getkd(double u, double v)` method, instead of returning `m_kd`, we return the texture colour with the (u,v) coordinates pass to the method.

   The UV coordinate is part of the `Intersection` class returned from the intersection test and it is calculated at the same time as calculating the intersection point.

4. **Bump Mapping**
   Relevant code is located in `Texture.hpp`, `Texture.cpp`, `PhoneMaterial.hpp`, `PhoneMaterial.cpp`, `Primitives.hpp`, `Primitives.cpp`, and `scene_lua.cpp`.

   Bump Mapping is very similar to texture mapping except that I modulate the object's surface normal rather than the colour, with this in mind, I reused the Texture class to store the bump map (normal map). Same as texture mapping, if a path to a normal map is passed to the constructor of the material, the material will have a `Texture *m_normal` field that contains the normal map information. Each time after the intersection test, if the material has a bump map, the modulated normal can be retrieved from the normal map with the UV coordinates by calling `normalMap(double u, double v, vec3 &normal)` defined in the Phong material class.

5. **Perlin Noise**
   Relevant code is located in `PerlinNoise.hpp`, `PerlinNoise.cpp`, `PhoneMaterial.hpp`, `PhoneMaterial.cpp` and `scene_lua.cpp`.

   I modify the `gr.material` command to take in another parameter as the scale of perlin noise. In the Phong Materal class, if the scale of perlin noise is not 0, the `bump(normal, point)` can be called to add perlin noise at the given point with the given normal. The noisiness is depend on the scale passed to the constructor.
   The noise function in Perlin Noise class is taken from Ken Perlin's website. Ken Perlin provides a Java implementation of Perlin Noise class and I used a C++ version of it with syntax translated directly from Java to C++.

6. **Photon Mapping**
   Relevant code is located in `PhotonMap.hpp`, `PhotonMap.cpp` and `A5.cpp`

I had a full photon mapping implementation with one photon map. I used the photon map class provided in Henrik Wann Jensen's book to accomplished this. When the program starts, it will start `photonMapping()` in `A5.cpp` which is basically a two passes algorithm:

```
PhotonMapping()
    PhotonTracing()
    RenderUsingPhotonMap()
```

In the first pass, a lighting simulation is performed by tracing photons from light sources and storing these photons as they scatter within the scene. Intuitively, photon mapping works by splitting the energy emitted by each light source into discrete packets called photons. After this process is done, a photon map will be constructed, which can be used to efficiently query lighting information.

1. Photon Emission: A number of photons are emitted from light sources and each photon represents a fraction of the total power of the light. Each photon include a position, direction and an associated power. and traced through the scene just as rays are in ray tracing. I used three random numbers to determine the emit direction.
2. Photon Scattering: After a photon is generate for emission, it is traced through the scene. This process proceeds analogously to tracing rays. When a photon intersects a surface it is either scattered, reflected, transmitted or absorbed. Russian roulette is used to choose which of these events occurs, the surface material properties determine the corresponding probabilities.
3. Photon Storage: Whenever a photon hits a non-specular surfaces, it is stored in photon map, otherwise, just keep tracing the photon with new position and direction. The photon map used a balanced kd-tree acceleration structure in order to perform these searches efficient.

```
PhotonTracing()
    num_emitted = 0
    while (not enough photons) {
        do {
            x = random number in [-1, 1]
            y = random number in [-1, 1]
            z = random number in [-1, 1]
        } while (x^2 + y^2 + z^2 > 1)

        d = <x, y, z>
        p = light source position
        tracePhoton(p, d)
        num_emitted = num_emitted + 1
    }
    scale power of stored photons with 1/num_emitted
    balance the photon map
```

The second pass is rendering the scene using the photon map from the first pass. For each pixel, I generate a ray through it from the eye position and when it hits a diffuse surface, I estimate the irradiance at the intersection point. The irradiance estimation is done by finding the k-nearest neighbors within a given range. The kd-tree structure in the photon map helps with finding these neighbors efficiently. Last, the irradiance returned is the colour of that pixel.

7. **Caustics**

Relevant code is located in `A5.cpp`

To simulate this with the full photon mapping implementation, I just allowed the photons to make any number of specular bounces, but at the first diffuse bounce, it is stored in the photon map and another photon is generated.

8. **Diffuse Interreflection**
   Relevant code is located in `A5.cpp`

   To simulate this with the full photon mapping implementation, just like for caustics, photons are stored at each diffuse bounce, but are still allowed to continue on their path, until they get absorbed.

9. **Area Light Source**
   Relevant code is located in `A5.cpp`

   Area light source is the easiest objective, when emitting the photon, instead of setting the emitting position to the light source, use two additional random number to generate a new random position around the point light source.

10. **Soft Shadow**
    The scene has the soft shadow effect after I implemented a full photon mapping and the area light source.

11. **Refraction**
    Relevant code is located in `PhoneMaterial.hpp`, `PhoneMaterial.cpp`, `A5.cpp` and `scene_lua.cpp`

    I extended the `gr.material` command to take in a transparency coefficient and a refraction index as additional parameters. If the transparency coefficient is 0, then it is not transparent and the material will stay the same as before. If the transparency is a number between 0 and 1, then when a photon hits the material surface, it will have some chance to transmit (due to Russian roulette) and recursively issuing a secondary refracted ray following Snell's law. The recursive process is similar to reflection's implementation.

    The refraction index for air is set to 1.0 and I added two more materials: water will refraction index 1.33 and glass with refraction index 1.85.

12. **Adaptive Anti-aliasing**
    Relevant code is located in `A5.cpp`

    I implemented the adaptive method which operates in the following manner:
    First, each pixel is fully Raytraced with 4 rays through each corner of the pixel and return colours for these 4 rays. Following this, a method iterates over these colours and check if these colours differ from each other by more than a certain threshold. If yes, then I will ray trace this pixel again with 2x rays through random positions on this pixel. Otherwise, I will just take the average of each corner to return the colour.

    This way I can concentrate the computational power only on the pixels that need Anti-aliasing more than other pixels, and not waste precious computational time by sending multiple rays through pixels that do not need this.

13. **Multi-threading**
    Relevant code is located in `A5.cpp`

    I used the C++ library ¡thread¿ and ¡mutex¿ to achieve mutithreading. Given the number of threads for photon tracing and rendering, multiple threads will be created for both of these two passes. Some mutex are used when modifying global variables like the photon map.

    The second pass of photon mapping won't start until all the threads used in the first pass join the main thread.

    Run time with single thread:

real	2m27.327s
user	3m14.368s
sys	0m1.978s

    Run time with multiple thread:

real	2m16.219s
user	3m4.652s
sys	0m1.716s

New Files Added :

- `CSGNode.hpp` and `CSGNode.cpp`: Subclasses of Geometry Node and can be used to represent the union, intersection or difference of two given geometry nodes.

- `PhotonMap.hpp` and `PhotonMap.cpp`: Contains a Photon class and a photon_map structure. These two new files are used for construct the global photon map, finding the k-nearest photons and estimate irradiance of a given point. (by Henrik Wann Jensen)

- `PerlinNoise.hpp` and `PerlinNoise.cpp`: Contains a Perlin Noise class that can generate perlin noise and used for bump mapping. (by Ken Perlin)

- `Texture.hpp` and `Texture.cpp`: Used to store information for both texture mapping and bump mapping. Can retrieve colour for a given point which can either be treated as colour of a material or a normal of a bump map.

- `Settings.hpp` and `Settings.cpp`: A class used to stored all the global variables passed from command line such as number of threads to be used.

Bibliography :

- **Realistic Image Synthesis Using Photon Mapping**, Henrik Wann Jensen, AK Peters, 2011
- "CS488/688 Course Notes", Craig S. Kaplan, Department of Computer Graphics, University of Waterloo, Fall 2016.
- L. Bergman , H. Fuchs , E. Grant , S. Spach, "Image rendering by adaptive refinement", ACM SIGGRAPH Computer Graphics, v.20 n.4, p.29-37, Aug. 1986
- University of Dartmouth, The Photon Mapping Method
- the Codermind team, Ray Tracing Tutorial
- Ken Perlin, Perlin Noise implementation from Ken Perlin's website.

Objectives:

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

___ 1: **Extra Primitives:** Extend the modeling language and at least one extra primitive (Cylinder) is implemented.

___ 2: **Constructive Solid Geometry (CSG):** Primitives can be combined using Constructive Solid Geometry.

___ 3: **Texture Mapping:** Texture mapping has been implemented, and texture mapped primitives (such as lemon slice) are properly rendered.

___ 4: **Bump Mapping:** Bump mapping has been implemented and is applied on the table surface.

___ 5: **Photon Map:** Construct a photon map using path tracing, make a viewer for the photons.

___ 6: **Diffuse Interreflection:** Rendering of diffuse interreflection via the photon map.

___ 7: **Caustics:** Rendering caustics via photon map.

___ 8: **Adaptive Anti-aliasing:** Anti-aliasing is carried out on the scene to remove jaggies.

___ 9: **Soft Shadow:** Soft shadow effect is properly rendered.

___ 10: **Final Scene:** A final, unique scene is created demonstrate the features in the above objective lists.

Extra Objectives:

___ 11: **Perlin Noise:** Support Perlin Noise for bump mapping in addition to the original bump mapping.

___ 12: **Multi-threading:** Enable multi-threading for both two passes of photon mapping.

___ 13: **Refraction:** Refraction is implemented which will have the caustics effect around the transparent objects.

___ 14: **Reflection:** Reflection I implemented in A4 has bugs and I fixed it in A5.

___ 15: **Area Light Source:** Area light source is added to the scene.

A4 extra objective:

1: **Supersampling**

2: **Reflection** (has bugs)