CS 488 Project Documentation

Raytraced Terrarium

Jenny Lei

j37lei

20824513

December 6, 2022

Final Project

The goal of the final project was to extend the basic ray tracer to render a scene with effects like texture mapping, reflection, refraction, and caustics.

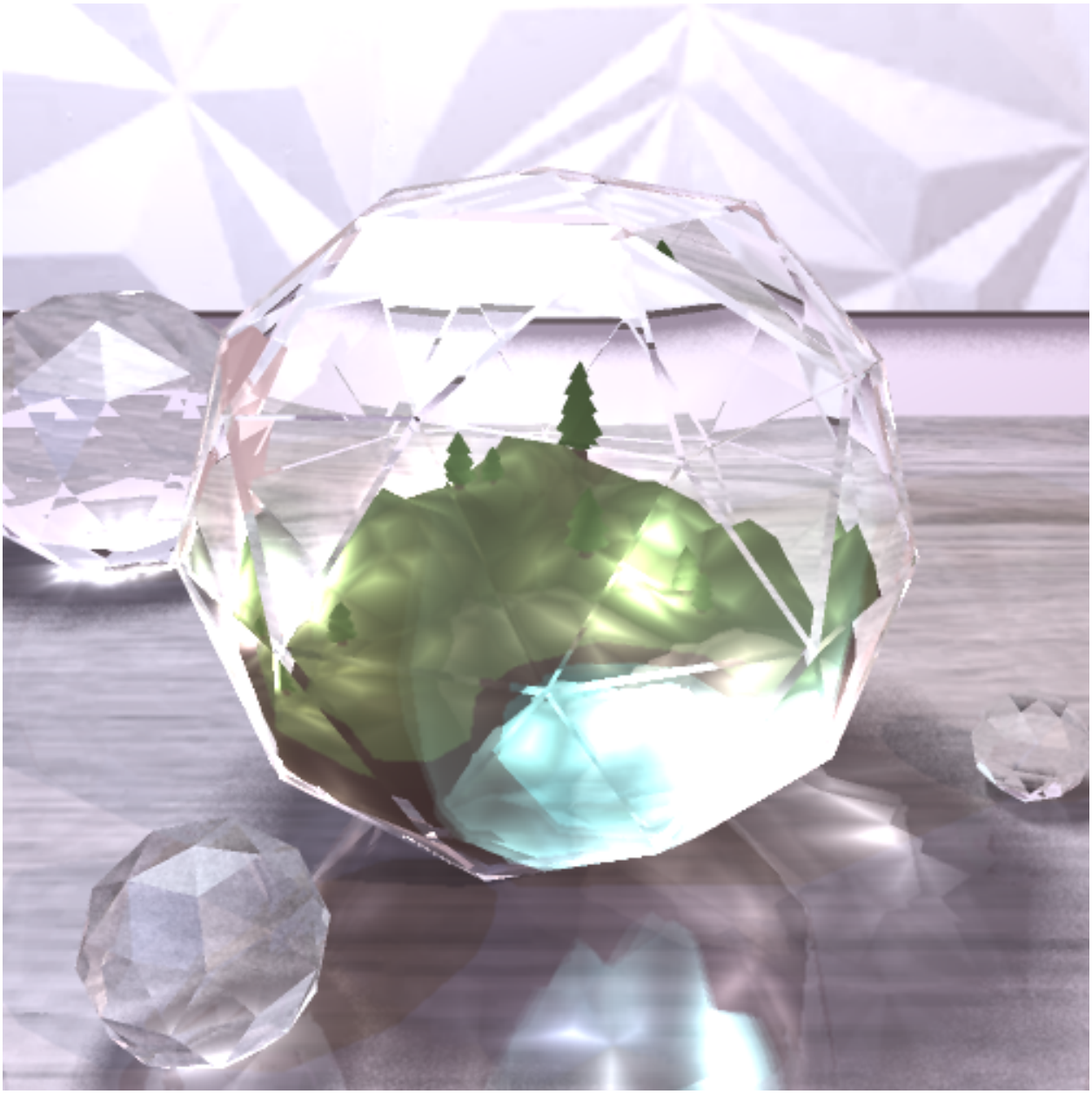


Figure 1: Final scene

In the scene is a glass polyhedron that has a miniature landscape inside of it. There is some water, grass, trees, and mountains in the glass polyhedron. It is placed on top of a table with a wall behind it.

The ray tracer needed reflection and refraction in order to properly render the glass polyhedron and the objects inside of

it. Furthermore, photon mapping for caustics was implemented to show the focusing of light as a result of the reflection and refraction. You can see the caustic effects in the shadows of the glass polyhedrons on the table.

Other effects include soft shadows of the glass on the table, the texture mapping of the table, and normal mapping of the wall. Adaptive anti-aliasing was also added to produce a smoother edges.

Organization

All files are located under the A5 directory.

The sub-directories are organized is this manner:

Assets/	contains all the asset files
Assets/Obj/	contains all the obj files
Assets/Textures/	contains all the texture and normal map png files
Demo/	contains all the images used in the documentation and demo docs
Demo/Renders/	contains additional renderings of the final scene
Docs/	contains all documentation files (README, proposal, documentation, and demo)
AntiAliaser/	contains the source files for the anti-aliasing classes
Light/	contains the source files for the light classes

Manual

To compile the source code (from the A5 directory):

```
premake4 gmake
make
```

To run the executable (from the A5 directory):

```
./A5 Assets/[lua file]
```

The rendered image will be located as defined in the lua file.

Implementation

Reflection (Objective 1)

I added mirror reflection by extending the lua scripts so that I can specify how much a material reflects in a lua file by doing

```
mat = gr.material(...)
mat:reflection([number between 0 and 1])
```

A reflection of 1 means that there is only reflection and 0 means there is no reflection (Figure 2).

During ray tracing, upon hitting a surface with non-zero reflection, cast a reflection ray from the point of intersection to get the reflected colour. The reflected ray direction is calculated using

$$R = I + 2 \times (-R \cdot N) \times N$$

where R, I, and N are the directions of the reflected and incident rays, and normal.

Thus, the colour of a point is obtain by doing

$$(1 - k_m)(L_a + L_d) + k_m L_m + L_s$$

$$\begin{aligned} L_a &= \text{Ambient colour} \\ L_d &= \text{Diffuse colour} \\ L_d &= \text{Specular colour} \\ L_m &= \text{Mirror reflection colour} \\ k_m &= \text{Amount of reflection} \end{aligned}$$

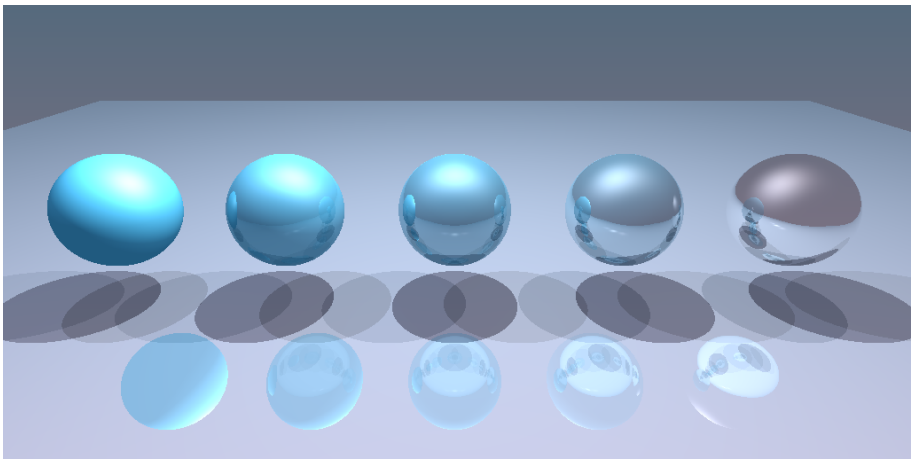


Figure 2: Different reflection values on spheres. From left-to-right: 0, 0.25, 0.5, 0.75, 1

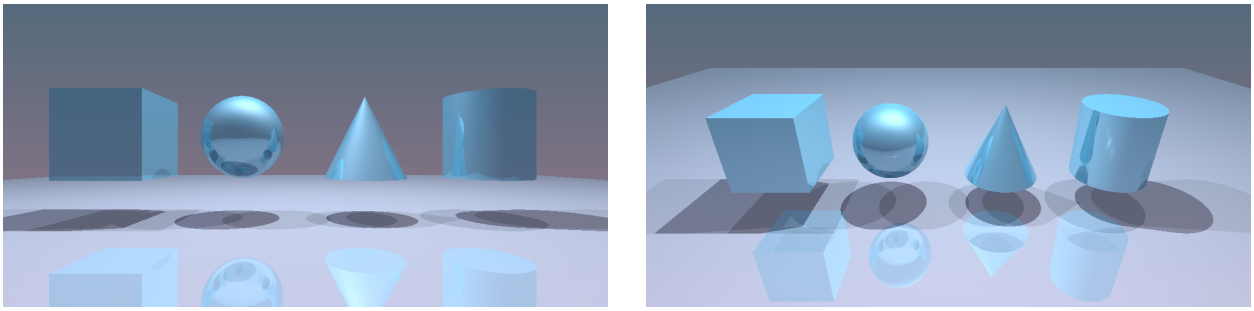


Figure 3: Reflection on different primitives

Refraction (Objective 2)

I added a property to each material for the material's index of refraction. The index of refraction was used to calculate the refracted ray direction, using Snell's Law, as well as the amount of refraction to reflection, using the Fresnel equations.

I also added another property to the materials class called transmission, which indicates what percentage of the material should be transmission/reflection and what percentage should be diffuse.

If we have

$$
\begin{aligned}
I &= \text{Direction of incident ray} \\
N &= \text{Direction of normal} \\
R &= \text{Direction of reflected ray} \\
T &= \text{Direction of transmitted ray} \\
\eta_1 &= \text{Index of refraction of first medium} \\
\eta_2 &= \text{Index of refraction of second medium} \\
\theta_i &= \text{Angle between the incident ray and the normal} \\
\theta_t &= \text{Angle between the transmitted ray and the flipped normal}
\end{aligned}
$$

Then the R and T are calculated as

$$
\begin{aligned}
R &= I + 2cos(\theta_i)N \\
T &= \frac{\eta_i}{\eta_2} i + \left(\frac{\eta_i}{\eta_2} cos(\theta_i) - \sqrt{1 - sin^2(\theta_t)} \right) N
\end{aligned}
$$

where

$$
\begin{aligned}
cos(\theta_i) &= -I \cdot N \\
sin^2(\theta_t) &= \left(\frac{\eta_1}{\eta_2} \right)^2 (1 - cos^2(\theta_i))
\end{aligned}
$$

Also, when $sin^2(\theta_t) > 1$, then total internal reflection occurs.

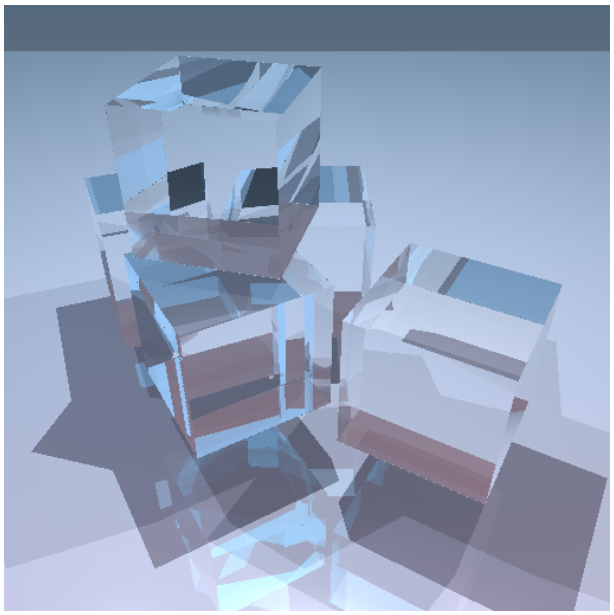


Figure 4: Refraction example on cubes

Let k_t be the amount of transmission, L_a, L_d, L_t, L_r, L_s be the colours caused by ambient, diffuse, transmission, reflection, and specular respectively. Then the colour at a point is

$$(1 - k_t)(L_a + L_d) + k_t(L_t + L_r) + L_s$$

The above equation can be combined with the one from mirror reflection (objective 1) to get

$$(1 - k_t)(1 - k_m)(L_a + L_d) + k_t(L_t + L_r) + k_m(L_m) + L_s$$

More Primitives (Objective 3)

I created new classes Cone and Cylinder which inherit from the Primitive base class to add support for cone and cylinder primitives. Figure 5 shows all the supported primitives.

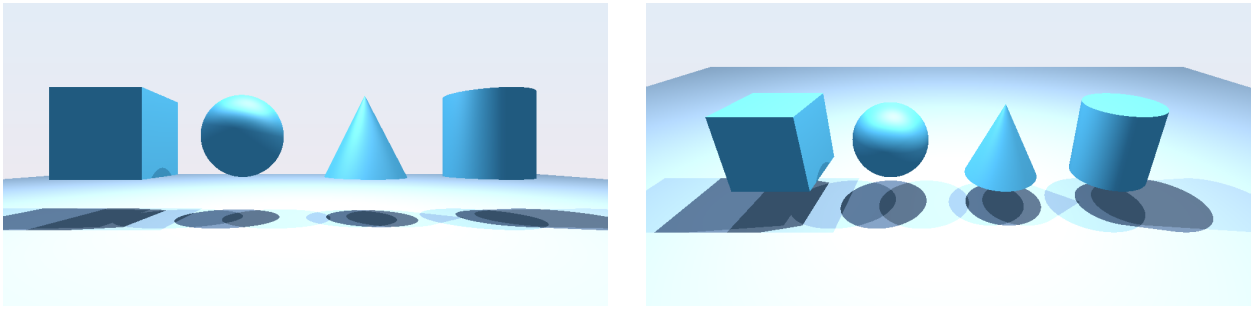


Figure 5: Ray tracer now supports cube, sphere, cone, and cylinder primitives

I used the implicit equations of cones and cylinders, substituted the equation of the ray, re-arranged the equation, and solved for roots of the equation. I also truncated the cones and cylinders so that they aren't unbounded.

Adaptive Anti-Aliasing (Objective 4)

For each pixel, I initially cast rays to each corner of the pixel. I then calculate the colour difference between adjacent corners. If the max colour difference is greater than some threshold, then subdivide the pixel by taking the middle point and recursively run the adaptive anti-aliasing algorithm on a smaller quadrant. If the colour difference is below the threshold, then no need to subdivide and just return the average colour between the four corners.

More specifically, define the colour difference between two colours A and B as the Euclidean distance between the two, so

$$\sqrt{(A.r - B.r)^2 + (A.g - B.g)^2 + (A.b - B.b)^2}$$

Figure 6 shows the first four calls to the adaptive anti-aliasing algorithm for a pixel. We only need to subdivide when we note that there is a large colour difference between sampled points in an area.

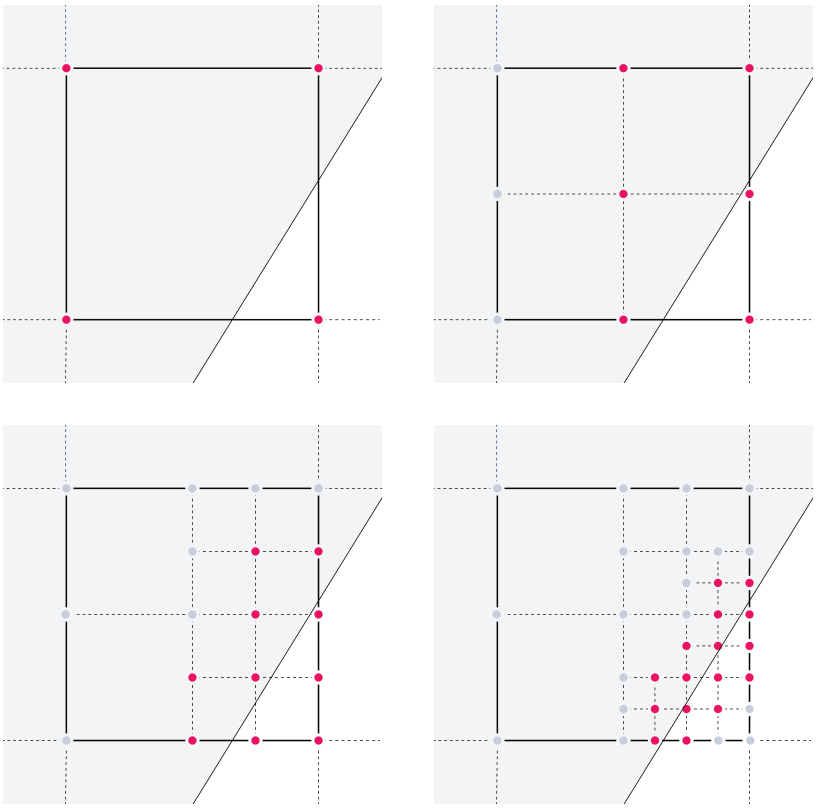


Figure 6: First four subdividisions for a pixel

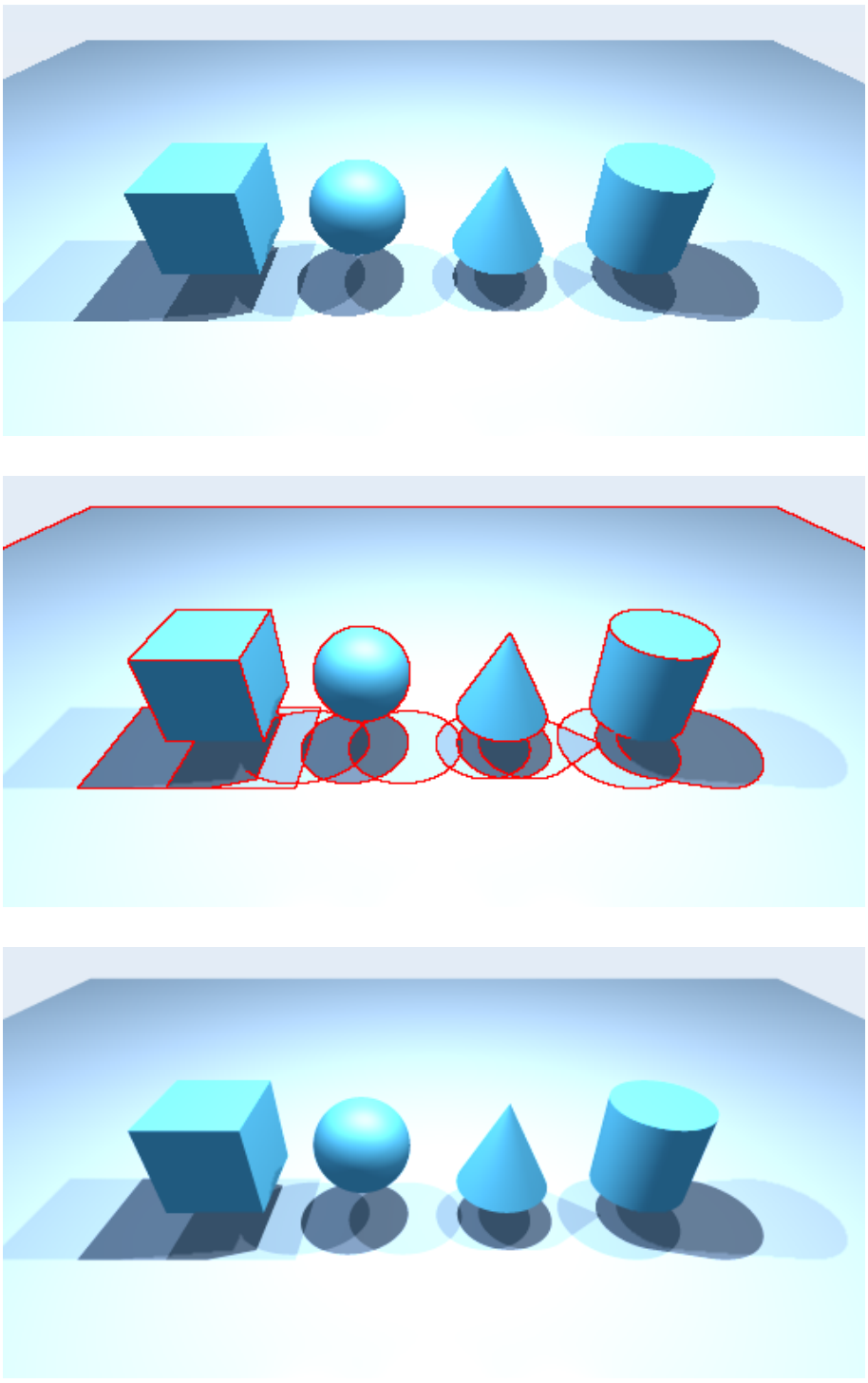


Figure 7: (top) no anti-aliasing, (middle) edge detection, (bottom) adaptive anti-aliasing

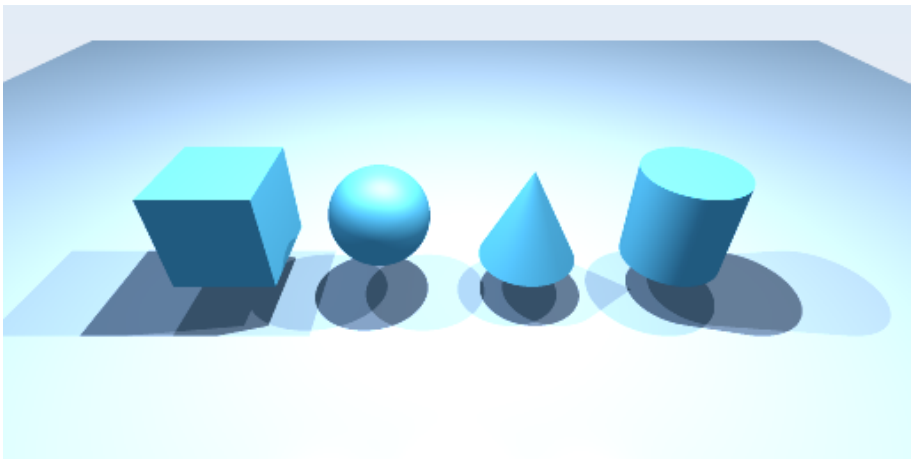


Figure 8: super-sampling 3x3 grid (extra-objective from A4)

While the rendered image from adaptive anti-aliasing (Figure 7 (bottom)) is similar in quality to super-sampling on a 3x3 grid (Figure 10), there is a large difference in terms of rendering time. The super-sampled render of the scene above took 123 seconds while the adaptive anti-aliasing only took 19 seconds to render.

Soft Shadows (Objective 5)

For soft shadows, I created separate classes for the various types of light sources, like point and spherical lights, which inherit from the original Light class. I added a virtual function to the base Light class that generates a list of sampled points in the light. For a point light, this function just returns a single value: the light's position. For a spherical light, this function returns a list of randomly sampled points that are in the spherical light. This can be found in the source files under the Light/ subdirectory.

Soft shadows are a result of a light source only being partially blocked by an object as shown in Figure 9

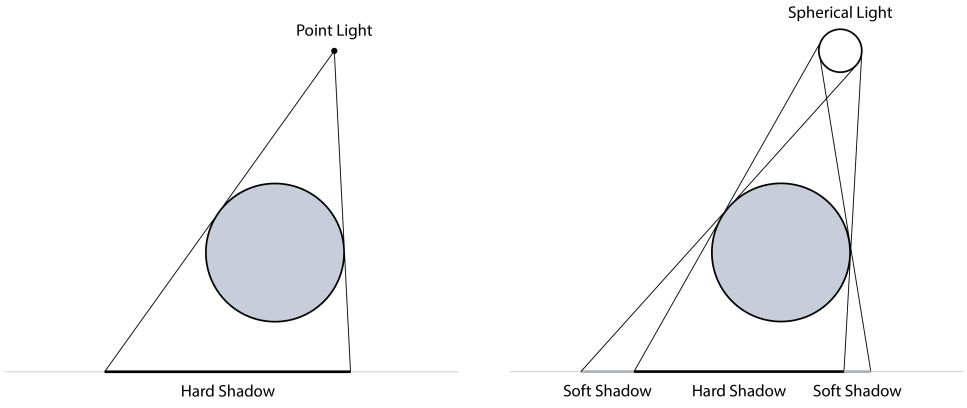


Figure 9: Hard shadow and soft shadows for point and spherical lights

During ray tracing, when doing shadow calculations between a point and a light, generate a list of light points and cast shadow rays to each of those points, counting the number of not obscured shadow rays (rays with no intersection). Thus,

for a point in the scene and a light source,

$$\%\text{ of light visible} = \frac{\#\text{ of not obscured shadow rays}}{\#\text{ of shadow rays casted}}$$

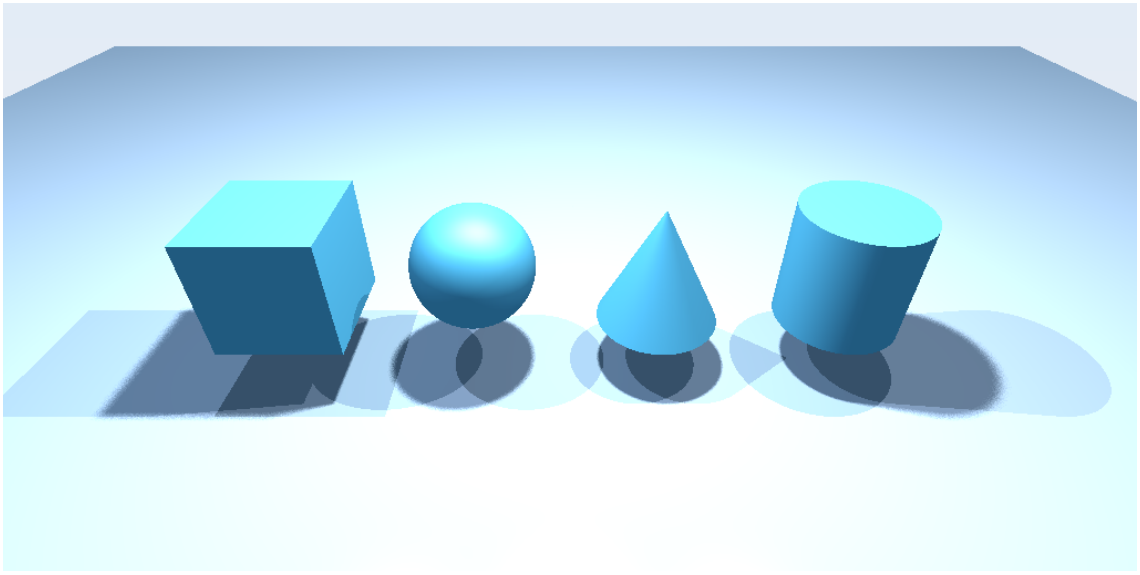


Figure 10: Soft shadows example

Texture Mapping (Objective 6)

I created a new base class called ImageMap, from which the classes TextureMap and BumpMap both inherit from. ImageMap contains a function to get the RGB colour at a point in a PNG image.

To texture map, I also added a virtual function to the base Primitive class. This function maps a point on a primitive to a (x, y) coordinate pair. This is only implemented for the primitive, non-mesh, objects. During the ray intersect primitive calculation, also keep track of which face was intersected to make the texture map coordinate mapping simpler.

During colour calculation, if an intersected object has a texture map applied, lookup the colour at the corresponding point in the texture map and use that colour as the diffuse coefficient instead of the material's initial diffuse coefficient.

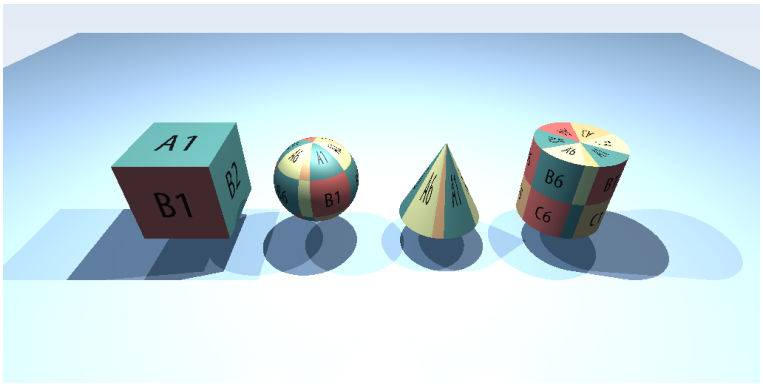


Figure 11: (left) texture map image, (right) texture mapped primitives

Normal Mapping (Objective 7)

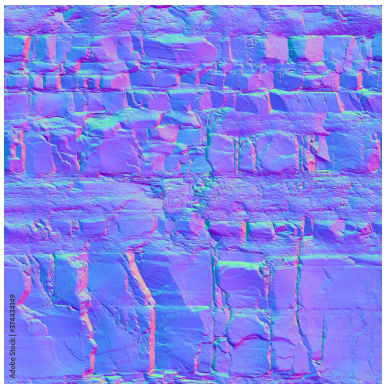
Similar to texture mapping, create a BumpMap class which inherits from the ImageMap base class. I use the same coordinate mapping from point of intersection to image to determine which point on the normal map to use. Once I get the normal map colour, I perturb the actual normal at the point of intersection based on this colour to get a new normal vector to use for colour calculations.

One thing to note is that the normal should be perturbed relative to its tangent plane whose axis are defined by the two orthogonal vectors u and v. u and v are obtained by taking the partial derivative of the surface at the point of intersection.

In order to take the partial derivative, first convert the surfaces implicit equation to a function of two variables, $f(x, y)$, instead of x, y, and z. This is done by letting $f(x, y) = z$ and isolating for z in the implicit equations. So now we have $f(x, y) = z =$ [some equation in terms of x and y]. Now we can take the partial derivatives of f with respect to x and y. Thus we have $u = normalize(x,\ 0,\ x\frac{\partial f}{\partial x})$ and $v = normalize(0,\ y,\ y\frac{\partial f}{\partial y})$.

Now, during the new normal calculation, if we have perturbation b and original normal N, then the new normal N' is

$$N' = u * b.x + v * b.y + N * b.z$$



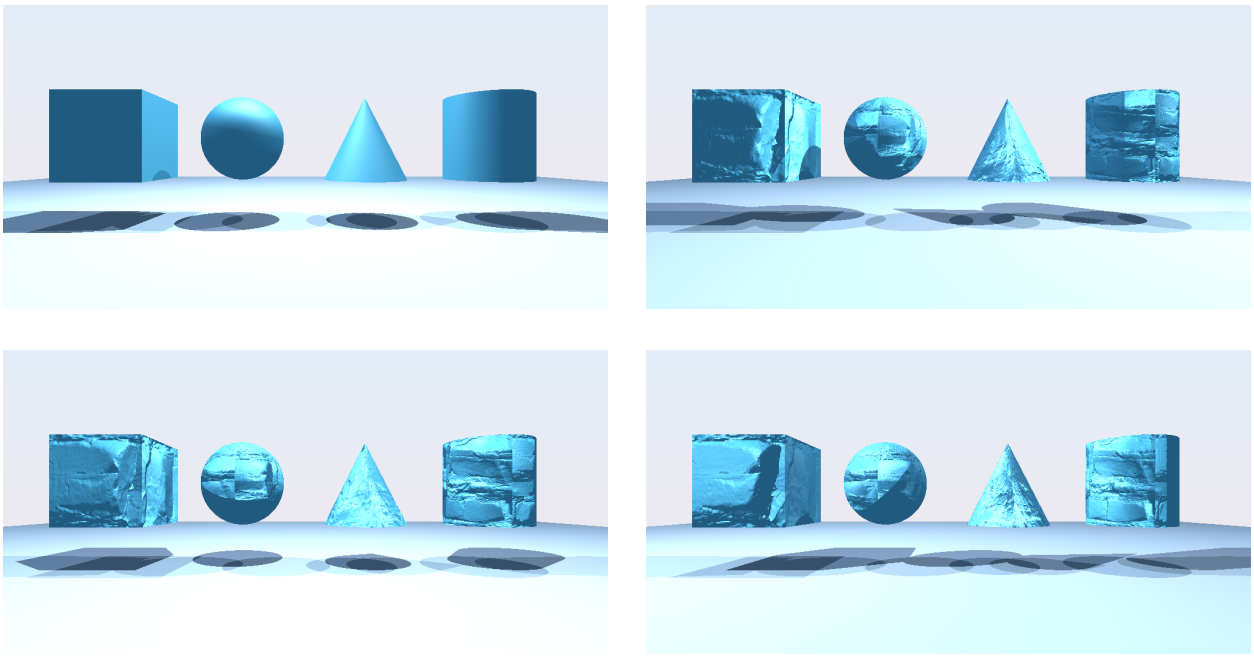


Figure 12: (top) normal map image, (right) normal mapped primitives with different light directions

Phong Shading (Objective 8)

When creating meshes, after loading in all the vertices and faces, I calculate the vertex normals of each vertex. The vertex normal is the normalized average of the surface normals of the faces that contain that vertex.

During a ray intersect mesh calculation, if a ray intersects a triangle of a mesh, lookup the vertex normals of each of the vertices in the triangle. Then use Barycentric interpolation to get the interpolated normal at the point of intersection.

To do Barycentric interpolation, convert the point to Barycentric coordinates, where a point P is represented as

$$P = uA + vB + wC$$

where A, B, C are the three vertex coordinates of the triangle and u, v, w are the Barycentric coodinates, $u + v + w = 1$, where

$$u = \frac{\Delta BCP_{area}}{\Delta ABC_{area}} \qquad v = \frac{\Delta CAP_{area}}{\Delta ABC_{area}} \qquad w = \frac{\Delta ABP_{area}}{\Delta ABC_{area}}$$

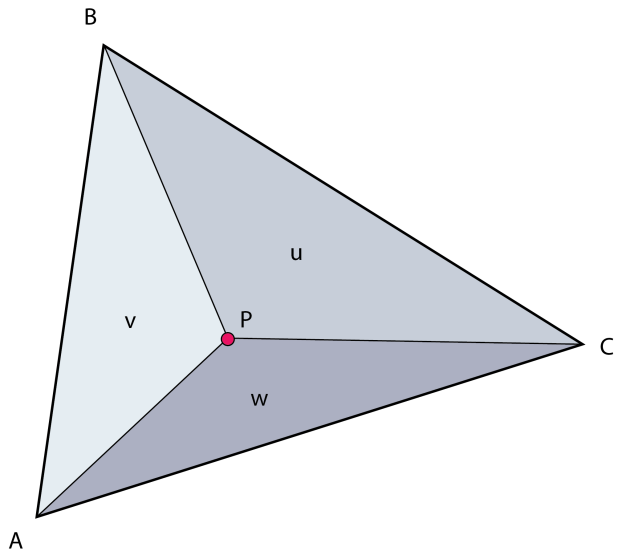


Figure 13: Subtriangles for Barycentric coordinates

Then the interpolated normal is just

$$N_P = uN_A + vN_B + wN_C$$

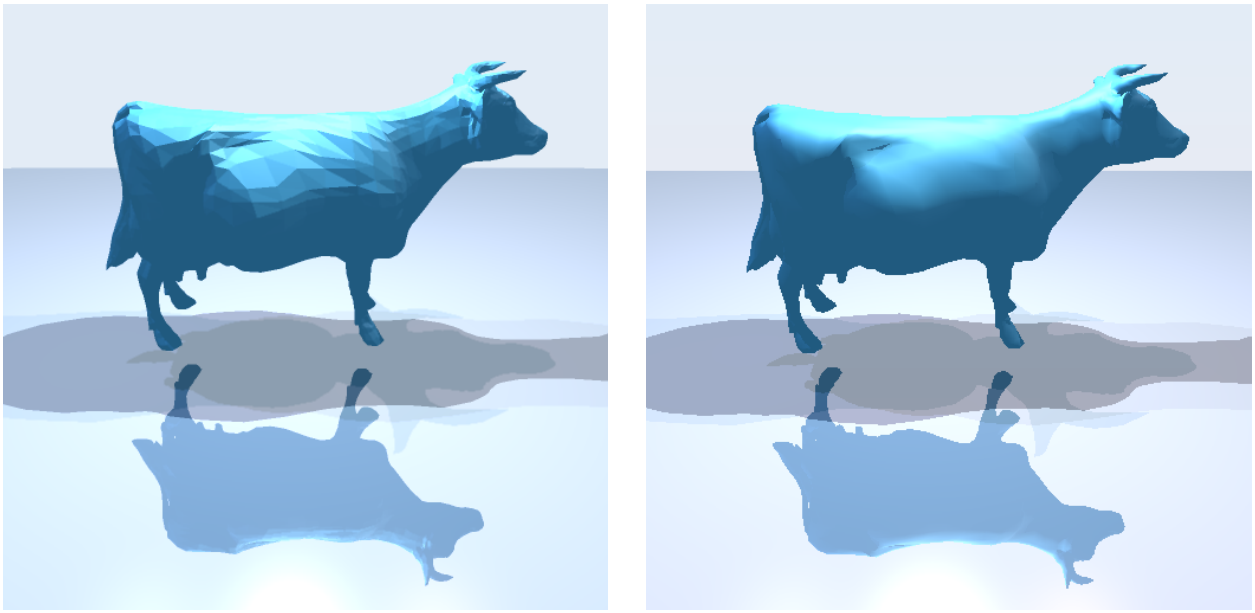


Figure 14: Phong shading example

Photon Mapping (Objective 9 & 10)

Photon Scattering

=== Summary ===

The idea is to build a caustic map containing photon information, like their positions and intensities, that can be used later during photon gathering.

In the CausticMap class, for each light, place a icosahedron mesh (20 sided polyhedron) around it. Then, for each face, cast 100,000 photon rays into the scene. Using Russian Roulette, trace the photon and store the photon information whenever it hits a diffuse surface. Then, once all the photons have been cast, build a kD-tree using all the photon informations for kNN searching during photon gathering.

=== Details ===

Random sampling of photon ray direction:

For each light and each icosahedron face, sample 100,000 random points within the face. For each sampled point, cast a photon ray from the light position through the sampled point into the scene.

For uniform random sampling of points within a triangle, ΔABC, generate two random numbers $r_1, r_2 \in [0, 1]$. The point on the triangle would then be

$$P = (1 - \sqrt{r_1})A + \sqrt{r_1}(1 - r_2)B + \sqrt{r_1}r_2C$$

Russian Roulette:

During photon tracing, instead of recursively casting multiple rays for diffuse reflection, specular reflection, and refraction, use Russian Roulette to choose between one of the three and absorption. If one of the three before are chosen, recursively cast and trace another photon.

The probabilities for diffuse reflection, specular reflection, and refraction are calculated using the material properties.

$$
\begin{aligned}
p_{\text{diffuse}} &= (1 - kt) \times \frac{kd_r + kd_g + kd_b}{3} \\
p_{\text{specular}} &= kt \times kr \\
p_{\text{refraction}} &= kt \times (1 - kr)
\end{aligned}
$$

where kd is the material's diffuse coefficients, kt is the amount of refraction / transmission, and kr is the portion of refraction that is reflected instead of refracted.

During Russian Roulette, scale the photons power by $\frac{\text{power}}{\text{probability}}$ when casting another ray.

kD-tree:

After obtaining a list of photons in the scene, construct a kD-tree using the photon positions. The code uses the c++ nanoflann library's kD-tree implementation. This kD-tree will be used during the photon gathering phase.

Photon Gathering

During ray tracing, when a ray intersects the scene, to determine the colour at that point, do the normal diffuse, specular, reflection, and refraction calculations. Additionally, add the caustic irradiance estimate to the normal colour to get caustic effects.

For caustic map irradiance estimation, complete a kNN search in the caustic map for the k nearest photons. Take a weighted sum of the neighbouring photon intensities such that the amount of caustic light at a point is

$$
L(x, \vec{w}) = \sum_p f_r(x, \vec{w}_p, \vec{w}) \frac{\Delta\Phi_p(x, \vec{w}_p)}{\pi r^2}
$$

where x is a point, $\vec{w}$ is the direction of the light, $\vec{w}_p$ is the incident direction of photon p, and r is the distance from x to its farthest neighbour. f is the BDRF and Φ is the flux of the photon.

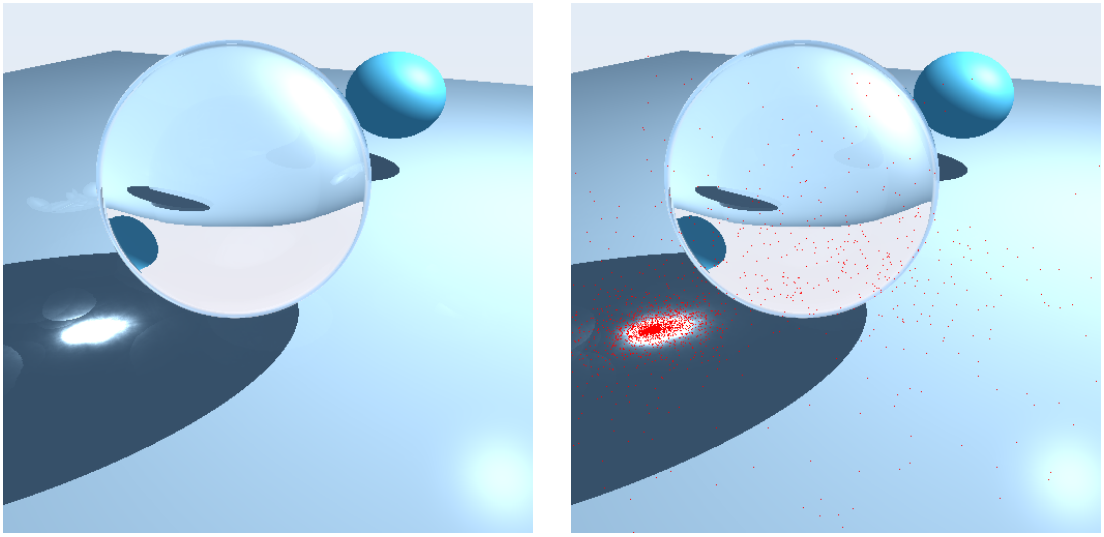


Figure 15: (left) sphere with caustic effects, (right) red dots are photons casted into the scene

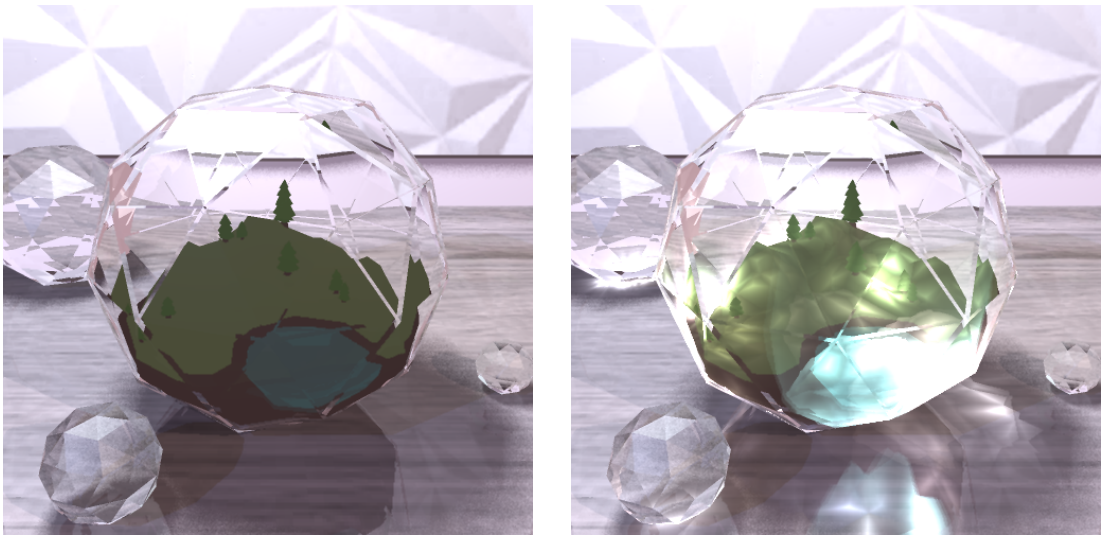


Figure 16: (left) final scene without caustic effects, (right) final scene with caustic effects

References

Blinn, J. F. (1978). Simulation of wrinkled surfaces. Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '78. https://doi.org/10.1145/800248.507101

De Greve, B. (2006). Reflections and Refractions in Ray Tracing.

Jensen, H. W. (2001). Realistic Image Synthesis Using Photon Mapping. AK Peters.

John, M. (2003). Focus on photon mapping. Premier Press.

Osada, R., Funkhouser, T., Chazelle, B., & Dobkin, D. (2002). Shape distributions. ACM Transactions on Graphics, 21(4), 807–832. https://doi.org/10.1145/571647.571648