Raytracer, Enhanced

Mingrui Zou

m2zou

20530377

December 4, 2017

Final Project:

Purpose :

To implement an enhanced version of the ray tracer from assignment 4 that will be able to render a more complex scene. I feel that I have completed all my objectives.

Statement :

The scene to be rendered by the enhanced ray tracer will be of a bathroom containing a bathtub.

The bathtub will be surrounded by walls on three sides and will rest upon a floor. The wall and floors will use a combination of noise generated textures, texture mapping, and bump mapping to create marble and textured tiles with lighting effects between the tiles from the bump mapping.

The bathtub itself will be filled with water. The water will have a refraction index that will bend light entering it. The water will have ripples that are generated procedurally. There will be caustics underneath the water due to the rippling above; this will be achieved via photon mapping.

There will be a rubber duck and a toy boat floating atop the water. These will be additional models that I will create for this scene.

The bathtub will have a faucet with a reflective metal surface. Other elements of the scene will be seen in the faucet.

Technical Outline :

Refraction

Refraction happens when the light passes from one medium to another medium, it changes the angle at which the light travels in relation to the normal of surface. The substance of the medium in regards to how well light can travel through them can be measured as $\eta = cv$ where c is the speed of light in a vacuum and v is the speed of light in the medium. Giving us η, the refraction index.

We can then use Snell's law, $\sin\theta_1 \sin\theta_2 = \eta_2\eta_1$, to compute the new angle at which light travels.

In the ray tracer, this means we can generate recursive rays using refractive surfaces, changing the angle at which we see the elements beyond the surface.

Reflection

We can generate recursive reflective rays by calculating the direction of a new ray using the surface's normal. The new ray can be calculated with $R = I - 2(N \cdot I)N$, where R is the direction of the reflection ray, I is the direction of the incident ray, and N is the surface normal. We can the intersection point of the incident ray and the surface as the origin point for the reflection ray.

Generated Water Ripples

We can generate water ripples by using sine waves to generate the normal of a surface at any point from the origin of the water ripples. At any distance from the origin of the water ripples we can calculate the normal as the tangent to the derivative of the sine wave. Using different sine functions and coefficients will result in varying ripples.

Photon Mapping

To achieve global illumination with shadows and caustics, we can generate a large number of rays (actual number to found through experimentation) from the light source and store the points where the rays hit a diffuse surface as a photon map. Since the photon map will be according to the world coordinates, we can store the photon map as a kD tree. We will also include additional information such as the surface normal and the intensity of the light.

When we ray trace, we can do a k-nearest neighbour search to get all the photon points around the point the ray hit. Then we can average the information from the photon points to get irradiance for the ray traced point.

Texture Mapping

We can use a bitmap image to create intricate designs for our surfaces. Using the ray traced point on the surface, we can get the location of the corresponding pixel or averaged pixel from the bitmap. This will be applied to the floor and/or wall.

Bump Mapping

This is similar to Texture mapping. But instead of taking the colour from the bitmap, we can interpret the RGB value of the pixel as a change to the normal at the ray traced point. This will be applied to the walls and floor to create a tiled aesthetic.

Marble Noise

We will be using random noise to generate a marble texture for the walls and/or floor. We can achieve this by generating a random bitmap of static. Then using linear interpolation and several zoomed-in versions of the same bitmap and combining them together, we can create a smooth looking marble texture.

Additional Models

Models of a rubber duck and a toy boat will be created for the scene. The models will be made using Blender and will be made from triangular meshes. They will be in the format of an .obj file, with a list of vertices and a list of faces.

Scene

The scene will composed together according to some aesthetic design.

Manual :

The build process is the same as A4. Actually, everything is still in the A4 directory, and nothing has been renamed.

premake4 gmake

make

./A4 Assets/bath.lua

High resolution images of the project are stored inside of the A4/Images/hi_res directory.

All other images from lua scripts will be saved to the Images directory automatically. There are a bunch of lua scripts in the Assets directory which I used to generate the demo images.

I do not guarantee that all the scripts will generate exactly the same image. This is mostly due to the photon mapping. Most of the demo images were ray traced without photon mapping turned on. To turn off photon mapping, you need to change the NUM_PHOTONS constant in Settings.hpp to 0. Then rebuild.

The scripts are:

refraction.lua

reflection.lua

water.lua

photon_map.lua
texture_map.lua
bump_map.lua
noise.lua
bath.lua

Modules :

For the sake of brevity, classes taken from the donated code of A4 will only describe what has been added to them.

A4.cpp:	Contains the main code for constructing primary rays and casting them.
Image.cpp:	Contains code for the saving and loading of png images.
Intersect.cpp:	Intersect class handles the storage of data collected from raycasting. It also handles the colour calculations of that intersection. Recursive ray casting happens in this class.
GeometryNode.cpp:	Handles reflective, transparency, and opaque properties. As well as the intersection checks against its primitive.
Mesh.cpp:	Contains code for ray intersection checking as well as uv coordinate calculation for meshes.
PhongMaterial.cpp:	Contains lighting calculations.
PhotonMap.cpp:	Contains all code for photon mapping and related data structures.
Primitive.cpp:	Contains code for ray intersection checking as well as uv coordinate calculation for basic shapes such as planes, spheres, and cubes.
Ray.cpp:	Very simple classes that determines a point and a direction. Most of the code for handling a Ray object is elsewhere.
SceneNode.cpp:	Handles the ray casting with transformations.
Settings.hpp:	Contains a lot of preprocessor variables.
Texture.cpp:	Contains classes for noise generation, wave generation, texture mapping, and bump mapping.

Lua Commands :

debug_render:	Functions similarly to render, except it takes an additional x and y coordinate. It will only shoot one ray from the pixel coordinate given.
plane:	constructs a one sided bounded plane geometry object.
bitmap:	Creates a Texture object for texture mapping given a png file. The png file must be placed inside of the Assets directory.
bumpmap:	Creates a Texture object for bump mapping given a png file. Sets the depth of the bump map given a number. The png file must be placed inside of the Assets directory.
ripple:	Creates a water ripple Texture object. Takes as parameters, the u location of the center, the v location, the width frequency of the waves, the height frequency of the waves, and the wave height.
noise:	Creates a noise Texture object. Takes as parameters, the base frequency of the noise along the width, and the base frequency along the height.
GeometryNode commands	:
set_bitmap:	Given a Texture object, sets the Texture object to be used when calculating colour.
set_bumpmap:	Given a Texture object, sets the Texture object to be used when calculating normals.
set_transparency:	Sets how transparent the object is from 0 to 1, then sets the refraction index.
set_diffuse:	Sets how opaque the object is from 0 to 1.
set_reflectiveness:	Sets how reflective the object is from 0 to 1.

Implementation :

Since most a good portion of my project involves modifying the properties of a surface given uv coordinates. I implemented a general Texture class that all other texture related classes inherit from. Then I added Texture members to the GeometryNode class so that each object can have a unique colour and normal modifiers. There are lua commands for creating textures and there are commands for setting the different textures of a GeometryNode

Texture mapping

Relevant code in Texture.hpp and Texture.cpp under the BitmapTexture class.

Texture mapping is implemented in the Bitmap class inside of Texture.cpp. When a ray is cast, it gathers information inside of the Intersect class. In addition to other information, we store the uv coordinates of that intersection.

The calculations of the uv coordinates is handled inside of each Primitive sub classes. For a sphere, we take the azimuth and the elevation as the uv. For a cube, each face has its own uv, so the texture is replicated six times. For a plane, we take the local x and y coordinates as the uv. For a mesh, we take the the uv of the intersection with the bounding box of the mesh.

Once we have the uv coordinates, we can get the colour value of the Bitmap class by mapping the uv coordinates, which range from 0 to 1, to the width and height value of the Image were using to hold the png file. The Bitmap class also handles tiling the texture, which is just a multiplication by the tiling value and then modulus to fit inside of 0 to 1.

Future considerations for texture mapping include better handling of uv coordinates. Being able to get good uv coordinates for the mesh and cube. Also, anti-aliasing for the textures through the use of precomputed lower resolution images, also known as mip-maps.

Bump mapping

Relevant code in Texture.hpp and Texture.cpp under the BumpmapTexture class.

Bump mapping works similarly to Texture mapping, which is why they both inherit from the Texture class. The way they use uv coordinates is the same. The difference is that instead of calculating grabbing the colour from the Image at a uv, the bump map grabs the change to the normal from the Image. However, grabbing the change to normal directly as an RGB value would make it cumbersome to create bump maps images. So instead, the BumpmapTexture class interpolates the change based on the height of the image, which is the averaged RGB value at a point.

The interpolation works by taking breaking up the image into a lattice. Then any given uv coordinate will fall between (inclusively) 4 height values. We then divide the square into an upper left triangle and a bottom right triangle. The uv coordinates then falls under one of two triangles. We then construct the triangles normal value by taking the cross product of two vectors in the triangle.

Once we have the normal value from the triangle, we have to change it to fit within the world space. We do this by passing in the actual normal of the surface we hit with the ray, as well as a vector that defines the upward direction relative to the normal. Using this information we can construct a change of basis matrix to change our bump normal to a usable normal in world space.

As with the texture mapping, the bump map can be tiled. The depth of the bump can also be modified, which is important because the height of the bump map depends on the scale of the object that the bump map is placed on.

Future considerations include a way to better interpolate height values. Like the texture mapping, there are some aliasing issues. Maybe we could break the height map into frequencies with fourier transforms and use the derivative at a point.

Noise Generation

Relevant code in Texture.hpp and Texture.cpp under NoiseTexture class

The calculation and handling of the NoiseTexture class is similar to texture mapping, where it uses uv coordinates. The main difference here is that NoiseTexture does not load a png file, but randomly generates an image.

First, NoiseTexture creates a vector of random colour values. Then it uses a hashing function to find the colour value based on the uv coordinate. This would produce static, so instead of having a random colour value for each uv, the colours are interpolated using smoothstep to allow for a smooth transition between a lower number of random colour points.

The final noise image produced is a fractal sum, which we get by layering multiple noise images on top of each other, adding the colour values together. With each layer, the frequency of the noise increases, while the amplitude drops. This creates a noise image that is not just static and not just blurry. The important thing to note here is that because we use a hashing function, we can use the same random values generated beforehand, which saves on memory.

Currently, the NoiseTexture class takes a height and width value which determines the frequency of the lowest layer. And it only generates black and white images.

Future considerations include making the class more robust, as there already exists code for coloured images. There are also variations to fractal sum that could be used. Also the inclusion of gradients. Noise could also be modified to create noisy bumps.

Water ripples

Relevant code in Texture.hpp and Texture.cpp in RippleTexture class

This is similar to bump mapping in how it handles uv coordinates. The difference here is that there is no png image to determine where heights are. The normal is calculated based on the derivative of a sine function. The sine function operates on the distance from the center of the ripple. We then calculate the normal as a tangent to the derivative of the sine wave. Then we transform that normal to world space according to the given surface normal and relative upwards vector, like with bump mapping. The result is a very smooth looking ripple as there are not issues of aliasing with calculating the normal directly from the derivative of the sine function.

Currently, the RippleTexture class can take changes to its central point, the frequency of the ripples in both uv directions, and the amplitude of the wave. The derivative of the base sine function is fixed at -sin(t), where t is the distance from the central point.

Future considerations involve using more complex functions to simulate waves getting further apart from one another as distance increases. Also including more than one point of rippling.

Refraction

Relevant code in Intersect.cpp, in the getLighting() method.

Refraction works by first setting the transparency and refraction index value of a geometry node. This tells the Intersect class, which handles information retrieved from a ray trace, how to calculate the lighting information from the surface. The surface normal, ray direction, and refraction index of the surface hit are used to calculate the angle at which the ray refracts. In some cases, it reflects due to total internal refraction.

Once the new ray is calculated, it is cast again out into the world where it gathers more information. The recursion happens in the getLighting() method, where it calls castRay on the SceneNode and then using the Intersect object returned from the cast, it calls getLighting to get the colour from the refracted ray. The recursion stops if MAX_DEPTH is reached, or if the colour contribution is lower than MIN_CONTRIBUTION. We pass the contribution value, how much it will change the overall colour of the pixel, to each recursion step and recalculate based on the transparency value of the object.

Reflection

Relevant code in Intersect.cpp, in the getLighting() method.

Reflection works very similarly to refraction, except that the calculation of the new ray to be cast is different. It uses only the normal of the surface hit and the direction of the ray to calculate the new ray. Everything else works like refraction, except that there is a reflectiveness value in GeometryNode that determines how reflective a surface is, instead of the transparency value.

Overall, there are three main contributors to the colour information, the transparency value, the reflectiveness value, and everything else. When we call getLighting() on an Intersect, we are getting a combination of reflectiveness, transparency, and diffuse. This is reflected in the code for GeometryNode, which contains the members for reflectiveness, transparency, and diffuse. These three values range from 0 to 1. When a ray intersects with a GeometryNode, it divides each value by the total so that they all sum up to 1 and those values are stored in the Intersect. This is so that reflectiveness, transparency, and diffuse can be easily manipulated in lua for a single object.

Photon Map

Relevant code inside PhotonMap.hpp and PhotonMap.cpp

We start with the implementation of the Photon class which will store the information for each emitted photon. This includes the flux of the photon, the location, and the incident direction.

Then we implement the emission of the photons. Given a number of photons and a list of lights, PhotonMap will calculate evenly spaced vectors in a sphere around each light. We do this via the fibonacci algorithm for generating points on a sphere.

The power of the lights is taken into consideration, but instead of just dividing the power of the lights over the number of photons for each light, we distribute the number of photons for each light according to their power level. This means that photons have roughly the same magnitude of flux.

For each photon, we cast a ray from the light source in the direction of one of the evenly distributed vectors from the light. The raycast functions mostly as normal, except for the handling of recursive ray casting properties. We use the russian roulette method for determining if a photon continues or sticks. So the flux of any one photon will always stay the same, but certain percentage of photons will stick and a certain percentage will go on. Given a large enough number of photons, this will converge to be the same as if we split the photon and its flux with each intersection. This saves us a lot of space when we go to store our photons.

When a photon hits a surface, it generates a random number, this is the probability we will use to determine where the photon will go. We have the probability of the photon going through, reflecting, or doing something with a diffuse surface from the transparency, reflectiveness, and diffuse values in the Intersect object from the ray cast.

In the case of transparency, we get the refracted direction and send the photon along. In the case of specular reflection, we get the reflected direction and send the photon along. In the case of hitting a diffuse surface, we either store the photon at that location or we calculate a random direction whose dot product with the surface normal is positive, and we send it along. Previously, errors were made when generating this random direction, as it was biased towards the poles of a sphere in world space. That has since been fixed, and the photon distribution is no longer biased towards particular directions in world space.

We want to make sure that photons are only stored at diffuse surfaces, so as long as the diffuse value of a GeometryNode is not zero, photons can be stored there. We also want to make sure that photons are not stored on the first place they hit, so we use the value of MIN_PHOTON_BOUNCE in Settings.hpp to determine how many times at least the photon must change direction. This is to model after real world global illumination.

We handle photons differently if they are considered for caustics. A photon is considered for caustics if it has never bounced off a diffuse surface, only reflective and transparent surfaces. As in, photons that come directly from the light source and land on the first available diffuse surface. We dont force the photon to bounce off the diffuse surface in this case, but it still has the random probability to bounce off anyways.

Now that all the photons have bounced around the scene sufficiently, we must store them in a data structure. Actually, we use two of the same data structure, one for global illumination and one for caustics. This is for efficiency as well as isolating particular effects.

The class PhotonTree is a kd-tree in 3 dimensions. It takes a std::vector of photons which it will store and use. It builds the tree recursively by taking the midpoint of a partition of the std::vector of photons according to the current dividing plane, which is either x, y, z aligned. Then it creates its two children in the same way, except using the next dividing plane. Recursing until it hits a leaf node, where the partition is a size of one.

Each PhotonNode stores pointers to each children nodes. If both pointers are NULL, then it is a leaf node. It also stores a pointer to the photon that acts as the split information. When we are building the tree, we also store the bounding box information for each node.

Once the kd-tree is built it can be used to search for its nearest neighbours. We especially want to be able to specify a maximum range for which we will search for the neighbours. Because each node is essentially a bounding box of all its children, we can determine if there is any possibility that that node could contain a point within range. We can then eliminate entire nodes from our search this way. And we only check leaf nodes which are inside a bounding box that is within range.

To add the functionality to search for the nearest k neighbours in the tree is simple. We do a search with our maximum range, and then we return the k closest points from our result. There is a better way to approach it where you actually use the distances of the gather neighbours so far to eliminate nodes, but I tried to do that and there were bugs so I opted to go with this method instead. In the end, it doesnt cost much computationally, since the search range is relatively small.

Photon gathering occurs in two steps, one for the global illumination and one for the caustics. The irradiance estimate is gathered at an ray intersection point by taking the PHOTON_GATHER_NUM nearest photons and summing their flux contribution together then dividing that by the area of circle, where the radius is the furthest photon from the point. For each photon, their flux contribution is multiplied by a bidirectional reflectance distribution function, which in our case is just the dot product of the photons incident vector with the surface normal, clamped from 0 to 1.

For caustics, the flux contribution is also scaled proportionally with how close the photon is to the point, this is to get sharper caustics. The closer the photon, the more it contributes. We use the value CAUSTIC_SHARPNESS to calculate the contribution. This does mean that a higher CAUSTIC_SHARPNESS will result in more overall irradiance for caustics.

The irradiance value is then used with the colour of the material and texture to output a final colour.

For future considerations, definitely make the photon map classes more robust. Currently re-emits photons for each render, which is not necessary if the geometry of the world hasnt changed. Having the option to specify photon map properties at runtime instead of with global constants would have been good. The bidirectional reflectance distribution function could also be different depending on different surface materials.

Model

The duck and boat model were both made in Blender. I had no previous experience with 3D modelling software, as you can probably tell. The polygons had to be triangulated before exporting to an .obj file. Also, for some reason, one of the triangles on the boat was inside out, which was fixed by reversing the direction of the vertices. There are also vertex normals, but I didnt implement Phong shading.

For future consideration. Implement Phong shading.

Scene

The final scene is rendered with Assets/bath.lua

It makes progressively larger renders of the image.

The walls use random noise to simulate marble. They are slightly reflective because someone recently polished the walls. The walls also have a tiled bump map to make the wall look tiled.

The floor is a texture mapped hardwood floor, which is a terrible idea for a bathroom.

The bathtub is made from planes and is also slightly reflective.

The faucet and knobs are made from cubes and sphere and are also reflective.

The water has an oversized yellow duck creating water ripples from the center. The water is transparent with a refraction index of 1.3 and is very slightly reflective. Photon mapping is used to create the water caustics, which look too regular due to the regularity of the water ripples.

There is a toy boat in the back, because the duck takes up too much space.

Version Control

I used git for version control. The repo is on my student account at git.uwaterloo.ca.

Acknowledgements :

https://www.scratchapixel.com/

This website was immensely helpful. It has (almost) everything. It is very in depth about the topics that it covers.

https://web.cs.wpi.edu/ emmanuel/courses/cs563/write$_{u}ps/zackw/photon_{m}apping/PhotonMapping.html$

Thanks to Zack Waters for this overview on how photon maps work.

General thanks to stack overflow.

Bibliography :

Reflection and Refraction

Introduction to Shading. Introduction to Shading (Reflection, Refraction and Fresnel), 15 Aug. 2014, http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel.

Noise Generation for Marble

Vandevenne, Lode. Lode's Computer Graphics Tutorial. Texture Generation Using Random Noise, 2004, http://lodev.org/cgtutor/randomnoise.html.

Objectives:

Full UserID:____________________________ **Student ID:**____________________________

___ 1: **Refraction:** The ray tracer is able to create transparent surfaces with varying refraction indices.

___ 2: **Reflection:** The ray tracer is able create reflective surfaces.

___ 3: **Generated Water Ripples:** Procedurally generated ripples for water surfaces is implemented.

___ 4: **Photon Mapping: Forward Map:** The ray tracer is able to generate a photon map.

___ 5: **Photon Mapping: Photon Gather:** The ray tracer is able to use a generated photon map to perform lighting calculations.

___ 6: **Texture Mapping:** Texture mapping is implemented.

___ 7: **Bump Mapping:** Bump mapping is implemented.

___ 8: **Marble Noise:** Procedurally generated noise for surfaces is implemented.

___ 9: **Additional Models:** A model of a rubber duck and a toy boat created for placement in the scene

___ 10: **Scene:** A realistic depiction of a modern bathtub with bathtub accessories surrounded by marble-tiled walls and floor is created.