Command Line Interface (CLI)

We implemented a powerful CLI interface to allow the user to heavily customize what the raytracer. This was important when we were testing because we always wanted the raytracer to render as quickly as possible, but when rendering out final images, we wanted the raytracer to render an image that looked really good. Thus, we created a CLI so that we could selectively disable or tune features. Running ./raytracer --help, will show you all the possible configurations.

Features

We implemented all the part 1 features, shadows and reflection, then 6 advanced raytracing features: antialiasing, depth of field, multithreading, glossy reflections, general quadratic surfaces, and refraction.

Antialiasing

We implemented adaptive anti-aliasing [3]. For each pixel, a ray is sent out of each corner of the pixel, and if all the rays come back coloured very similarly (ie. a similar RGB colour), then we shade that pixel the average of the four rays. If the rays colours are different, then we randomly send more rays (specified by the CLI, and we typically 32 rays), inside the pixel and average those. This allows us to only spend more computation time where the colour varies greatly, like at the edge of objects. In addition, since the corner of each pixel is not unique to a pixel, that colour value can be saved and used later when we need the corner of the pixel to the left, right, above or below it. Thus instead of spawning 4 * pixels number of rays, we only need to spawn one ray per pixel, because we have already spawned a ray from 3 of 4 corner points. Unfortunately since the each row gets its own thread, we can only save the rays for use for the pixel to the right. Still, this provides a 2x increase over not saving any corner colour values, as we only spawn two corner rays per pixel and get the other colour values from the previous pixel. In all our rendered scenes we have enabled antialiasing, so all images serve as a demonstration for this feature.

Depth of Field

We implemented depth of field, by spawning multiple rays per pixel. All the rays per pixel (for depth of field purposes) go through a single point on the focal plane, but start from a different origin point. The origin point is a random location slightly deviates from the origin, (0, 0, 0). The focal plane is the plane where all objects remain in focus, and as you get farther from the focal plane the blurrier objects will be. The aperture size is the distance from the origin that rays should spawn out of. It is analogous to the size of the camera lens. The location of the focal plane, the aperture size, and the number of rays per pixel are all configure through the CLI. In the example image, depth-of-field.png, rendered from scene #3 in our code, we demonstrate this feature. As we can see, the middle sphere is aligned with the focal plane, and thus is not blurred at all. The sphere on the right is the farthest from the focal plane, thus it is the most blurred.

Multithreading

We implemented multithreading using OpenMP [4]. Each row in the rendered image will be on a separate thread, and there are as many threads as there are processing cores on the machine. Since each pixel in the entire image is independent (ie. you can shade each pixel without knowing any information from the other pixels), multithreading is very useful in a ray tracer. We could have made a thread for each pixel, but we chose to have a thread per row, because typically a computer will not have more cores than the number of rows in an image, so the additional benefit of giving a thread per pixel is zero. Furthermore, the benefit would actually be negative, because of the set up involved in making each thread, and the context switching that would need to constantly be done. Multithreading allows a huge increase in processing speed, proportional to the number of cores you have.

Glossy Reflections

We implemented glossy reflections by spawning multiple rays when we encounter a reflection.

The rays are shot out slightly deviates from the perfect reflection direction, and the distance from the random ray to the perfect reflection distance is determined by the material property, ‘glossiness’. There are two types of reflections, thus both must be made glossy: specular reflections and reflections off objects. The number of rays for each reflection is configurable through the CLI. Shown in glossy-reflections.png, rendered from scene #4, we have three spheres, where the right-most sphere is glossy, and the others are not. As we can see, the glossy sphere’s reflections are much blurrier as a glossy material would appear in real life.

Unit Sphere and Unit Square

The prototype for these two objects was present in the sample code provided, but the intersection code for both of them had to be implemented. Both of these implementations were relatively straight forward and issues only arose when shadows, reflections and refractions were implemented (discussed below) in addition to these.

General Quadratic Shapes

Any general quadratic surface can be expressed in the following form:

$$f(x, y, z) = ax^2 + by^2 + cz^2 + dyz + ezx + fxy + gx + hy + iz + j = 0$$

Therefore, given the 10 input parameters a through j, the intersection with any quadratic implicit surface can be determined. The math required to compute the intersection with a general quadratic surface is very similar to that of a sphere, but more generalised. [1] By default, these objects are centered at the origin.

In scene 6, rendered as quadratic-surfaces.png, we see several shapes demonstrated through the use of this one GeneralQuadratic object. More specifically, it shows a sphere, a cone, a cylinder, an elliptic paraboloid, and a hyperboloid of one sheet. Other common general quadratic shapes which can be represented using the same object include hyperboloid of two sheets, hyperbolic paraboloid, and many more...

Shadows

Shadows were a very straight forward feature to implement. At every single intersection, an additional ray is shot from the point of intersection towards the source of light currently in context. If there is an object blocking the way between the intersection point and the light, then the shadow ray will return a collision within the appropriate bounds (a t_value greater than 0 and less than or equal to 1). If this is indeed the case, then the specular and diffuse components for that light will be omitted and only the ambient component of the phong model will be accounted for. Otherwise, normal phong shading is used.

There was an issue where shadow rays were colliding with the object on which they were originating. However, this was accounted for and discussed later in this report.

Shadows were integrated into most of the scenes being submitted and can thus be observed in multiple places.

Reflection

Reflection rays are generated for every single collision if the material that the ray collided with is reflective. A new ray is created starting at the intersection point, directed in the reflecting direction, and shot out to determine what the reflection colour is.

There are two values that every material has associated with reflection properties (if it is reflective): reflection and reflective damping. The first parameter is a value between 0 and 1 that is multiplied by the reflection colour after it is computed to determine what fraction of the reflective colour should be applied to the material. The second parameter is used to decrease the effect that reflection has as distance increased. Reflective damping is used to specify the rate at which reflection will be decreasing as distance increases. It follows an exponential monotonically decreasing function.

One of the parameters that must be provided when a scene is rendered with reflection enabled is the maximum reflection number. Consider a scene with two mirrors facing each other and the eye located in the middle between them facing one of the mirrors. A ray reflecting of a mirror will reflect onto the other mirror, and back, infinitely many times... To avoid this infinite ray reflection situation, a reflection number is attached to each reflection ray by incrementing the reflection number of the source ray.

Refraction

There are two states that must be accounted for when determining the refractive ray at some point of intersection: a ray entering a refractive medium and a ray exiting a refractive medium. This can be determined by taking the dot product of the incident direction and the outward facing normal at the surface. Whether or not the ray is entering or exiting the medium will slightly vary the calculations. [2]

If a ray collides with a refractive object, two refractive indices must be known in order to determine the direction in which the refraction ray should be travelling; the refractive index of the medium the ray was travelling in, and the medium that its entering. In order for a ray to know what medium the it is currently travelling in, it’s new medium is computed every single time a new refraction ray is calculated.

In addition to refraction ray calculations, the refraction calculations are also accompanied by reflectance calculations. If a certain material is both refractive and reflective, we need to determine what percentage of the light gets reflected and what gets refracted. Reflectance is the weighting property used to determine this exact property when an objective is both refractive and reflective. If an object is only reflective or only refractive, the color retrieved from the associated property is used in its entirety with no additional weighting.

Just like the reflection numbers discussed above, refraction numbers were also implemented in similar fashion for refracting rays.

Scene 5 contains a lot of small distant golden balls, 2 large glass balls, accompanied by grass/sky like background. The two glass balls have slightly different refractive indices which determine how much the light bends as it enters and exits the balls.

File Structure

scene_object.{h / cpp}

The file structure of these files wasn't changed very much. They define the shapes of 3 different types of objects that could be rendered in some scene: a unit square, a unit sphere, a general quadratic surface. It should be notes that all three of these objects can later be scaled and translate. It should also be noted that a unit sphere can be defined by a general quadratic surface.

These objects are only responsible for storing the shape of each object, and nothing associated with their lighting or material properties. Their primary function is to determine if a collision has occurred with some ray.

Object collisions with itself

While the math required to the define the surface of a plane, sphere or a general quadratic surface is relatively straight forward, the main issue encountered while writing the code in these two files occurred when shadows, reflections and refractions were implemented. For the three effects mentioned above, a new ray is spawned after a collision with a prior ray has already taken place. The origin point of this new ray is always at the intersection point of the original ray and the surface at hand. A common issue was encountered was when a shadow / reflection / refraction ray collided with it its start point, thereby causing unrealistic reflections and shadows. To fix this, each ray had to know on which object it originated from. In addition to that property, a small epsilon was used so as not to prevent objects from being able to reflect off themselves or cast shadows onto themselves to, while still avoiding unnecessary collisions.

light_source.{h / cpp}

We do the shading for all the components of the phong model in this file: ambient, diffuse and specular. Each light source contributes an ambient component, which is then averaged and applies to all rays, and each light source contributes a diffuse and specular component, which are added to each ray. If the material is glossy, the specular component requires more than 1 rays, which are all averaged.

All the logic required for computer the shadow, reflection and refraction rays is done here. As these are all factors that affect lighting properties at the surface of a material, we decided to move the logic for computing those rays to this class. If reflection, refraction or shadows are enabled, the raytracer makes use of this functions to retrieve the appropriate rays. Once the rays are retrieved, they traverse the scene and compute shading like all the other rays.

raytracer.{h / cpp}

This is the main ray tracing file. It is responsible for parsing the command line input, then setting up the scene with all the objects and their respective affine transformations. Then the scene is rendered by sending a ray through each pixel. Depending on what features are enabled, there may be more than one ray spawned per pixel.

util.{h / cpp}

Operators

Though a lot of operators and Matrix/Vector/Point/Colour operations already came with the provided code, there were a few additions made to simplify the code in various places.

Added properties to Material, Intersection and Ray3D

There were several properties added to the objects below in order to make the new features possible.

Material

- reflection: a ratio representing how much of the reflection+refraction colour returned should be added to the intersection point.
- ref_damping: The damping coefficient for the exponential function determining how fast reflection decreases as distance increases.
- n: the refractive index of the current materia.
- glossiness: This defines how glossy a material is. The larger this value the more randomly deviated the reflection ray will be from the perfect reflection direction.

Intersection

- sceneObject: A pointer to the scene object with which the ray intersected.

Ray3D

- reflectionNumber: Prevents the creation of an infinite number of reflection rays.
- refractionNumber: Prevents the creation of an infinite number of refraction rays.
- startObject: A pointer to the scene object on which the ray originated.
- currentMedium: A pointer to the scene object through which the ray is travelling (NULL if its a vacuum).
- currentMaterial: A pointer to the material through which the ray is travelling (NULL if its a vaccum)

Rendered Images

We rendered many images to demonstrate all of our features. Each image is listed below along with the command line parameters used to render it.

refractions.png: ./raytracer --width 1280 --height 720 --scene 5 --phong --reflection 2 --refraction 4 --shadows --antialias 32

depth-of-field.png: ./raytracer --width 512 --height 512 --phong --depth-of-field 200 0.3 6 --antialias 32 --scene 3 --shadows

quadratic-surfaces.png: ./raytracer --width 1280 --height 720 --phong --scene 6 --antialias 32 --reflection 10 --shadows

glossy-reflections.png: ./raytracer --width 512 --height 512 --scene 4 --phong --reflection 1 --glossy 100 --antialias 32 --shadows

Contributions

Amandeep Grewal (998151716): Phong shading, Depth of field, antialiasing, multithreading, glossy reflections
Daniel (998207159): Unit square/sphere, general quadratic surfaces, reflections, shadows, refraction

In addition, using BitBucket, we reviewed and approved each others code.

References

[1]. http://mrl.nyu.edu/~perlin/courses/fall2013/sep25b/
[2]. http://ray-tracer-concept.blogspot.ca/2011/12/refraction.html
[3]. http://paulbourke.net/miscellaneous/aliasing/
[4]. https://computing.llnl.gov/tutorials/openMP/