Raytracing

CS475/675 A3

August 28, 2016

We start our exploration of "Rendering" – the process of converting a high-level object-based description of scene into an image. We will do this by building a raytracer – a program that converts 3D geometry into 2D pixels by tracing light rays backwards through a scene. Raytracing simulates light rays moving through a scene, and can produce physically accurate representations, as explored in detail in CS775. In this course we will settle for an approximation of light transport through a scene by limiting and approximating the interactions that light can undergo.

Figure 1: Ray-traced image

0.1 Required Reading

If you need background reading beyond the lecture slides, please see Chapters 2 (Miscellaneous Math), 4 (Ray Tracing), 6 (Transformation Matrices) and 13 (More Ray Tracing) of Shirley and Marschner's "Fundamentals of Computer Graphics", 3rd ed.

1 Assignment Specifications

Write a program called `trace`. It will take two arguments – the input scene file, and the output image file name. The raytracer will revolve around a `traceRay` function that takes a ray as argument, and returns the color of that ray. This function will be used recursively.

1. Raytracing:
   - Render a scene by casting, for each pixel on the viewport, a ray from the eye through this pixel into the scene.
   - Find which, if any, object intersects with this ray.
   - Find the color of this ray by performing a shading calculation for the intersection point.
   - **Reflections:**
     - Create rays reflected off objects (the so-called "bounce ray") by reflecting the incoming ray around the normal.
     - Calculate the color by recursively calling `traceRay`, attenuate the color of this bounce ray by some material property m_r times the color of the reflective object.
     - Limit Bounce Depth: Terminate the recursive nature of raycasting after some specified depth (the optional third command line argument).
   - **Anti-aliasing using Distribution Raytracing:** Shooting only a single ray per pixel can result in objectionable aliasing effects (jagged edges); these can be reduced by shooting multiple rays per pixels and averaging the returned (r,g,b) intensities. The View class is already set up to generate multiple rays per pixel. The rays are sampled according to a regular grid over the pixel (i.e. 2, 3, or 4 samples per pixel edge), and then jittered to break up regular noise. The number of samples is set by the `rays_per_pixel_edge` parameter in the View class. **You do not have to do anything extra for this in 3(a), but you might need to in 3(b).**
   - **Falloff for Lights:**
     - We want to model point lights as dimming with distance, and be able to adjust the factor by which this extinction occurs.
     - The Light class stores a falloff and a dead-distance. To calculate the color of a point light (the only light affected by falloff), calculate a scaling factor consisting of: $\frac{1}{(\text{dist}+\text{dead_distance})^{\text{falloff}}}$, where *dist* is the distance between the query position and the light source, and scale the illumination of the light by this amount.

2. Transformations:
   - **Linear transforms on spheres:** Support rotate, translate, and non-uniform scaling transformations on the sphere. The scene parser will load these transforms from the scene file into the scene for you. All transforms are represented in homogeneous coordinates by 4x4 matrices.
   - **Ray transforming:** Intersection tests with a transformed sphere are done by inverse transforming the ray from world space to model space (find the function call to do this), then testing it for intersection with the untransformed sphere. Supporting this will allow you to easily render ellipsoids. Remember to make a copy of the ray before transforming it, so as not to affect the original ray passed to the `intersect()` function!
   - **Note on Spheres:** Since we support arbitrary transformations, all spheres can be considered the unit sphere at the origin, with some compound transformation applied to them!
   - **Transforming Normals:** However, the normal vector for the surface at the point of intersection is now calculated on the untransformed sphere, and needs to be transformed to be properly oriented in world space. Please see Shirley section 6.2.2 or `http://www.unknownroad.com/rtfm/graphics/rt_normals.html` for details on transforming normals.

3. Shading:
   - Calculate color according to the Phong reflection model. (See section 2)
   - Take each light into account by summing the shading calculation over all lights, using the light's direction vector to calculate the necessary angles.
   - Remember to handle shadows. (See section 2.4)
4. Directional Lights:
   - Directional lights are like the sun - "infinitely" far away light sources with light rays moving in a specific direction.
   - Model lights as emitting a color $\boldsymbol{c} = (r, g, b)$, with light rays (photons) moving in a direction $\vec{\boldsymbol{d}} = (x, y, z)$, allowing you to calculate the angles needed for shading.
5. Point Lights:
   - Point lights emit light from a specific position $\vec{\boldsymbol{p}}$.
   - Model lights as emitting a color $\boldsymbol{c} = (r, g, b)$, with light rays (photons) moving in all directions from $\vec{\boldsymbol{p}} = (x, y, z)$, allowing you to calculate the angles needed for shading.
6. Primitives and Material Properties:
   - Support a transformed unit sphere.
   - Each sphere has a color $\boldsymbol{c} = (r, g, b)$ where each component is in the range of $(0.0, 1.0)$.
   - Support the following material properties, each of which is a coefficient in the shading calculations: m_a, m_l, m_s, m_{sm}, m_{sp}

2 Shading - Phong Reflectance Model

For this assignment you will use a slightly different formulation of the Phong Reflectance Model seen in class to calculate the color of a pixel. Our formula for the Phong Reflectance Model, where $\boldsymbol{\rho}$ is the (r, g, b) intensity of light sent towards the eye/camera, is:

$$\begin{aligned}\boldsymbol{\rho} &= m_a \boldsymbol{CA} + \sum_{Lights} \Big(m_l \boldsymbol{CI} \max(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) \; + \; m_s \boldsymbol{SI} \max(-\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}, 0)^{m_{sp}} \Big) \\ \boldsymbol{S} &= (m_{sm})\boldsymbol{C} + (1 - m_{sm})(\mathbf{1}, \mathbf{1}, \mathbf{1})\end{aligned}$$

Note: This formulation separates the "color" of the object from the components in the Phong formulation. The equivalent parameters in the lecture slides are:

$$\begin{aligned}\boldsymbol{k_A} &= m_a \boldsymbol{C} \\ \boldsymbol{k_L} &= m_l \boldsymbol{C} \\ \boldsymbol{k_S} &= m_s \boldsymbol{S} \\ \sigma &= m_{sp}\end{aligned}$$

$\boldsymbol{I} = (r, g, b)$ is the intensity of the infalling light at the surface point (attenuated by falloff).
$\boldsymbol{A} = (r, g, b)$ is the intensity of the ambient light in the scene.
$\boldsymbol{C} = (r, g, b)$ is the "color" of the object.
$\boldsymbol{S}$ is the specular highlight color linearly interpolated according to m_{sm} above
$\hat{\mathbf{l}}$ is the incidence vector **from** the intersection point **to** the light.
$\hat{\mathbf{n}}$ is the outward surface normal vector.
$\hat{\mathbf{u}}$ is the vector along the (backwards) viewing ray, **from** the viewer **to** the intersection point.
$\hat{\mathbf{r}}$ is the reflectance vector, supplying the angle of reflected light.
m_a, m_l and m_s are the ambient, lambertian and specular surface reflection properties of the material.
m_{sm} is the metalness of the material.
Colors multiply component-wise, thus $\boldsymbol{CI} = \{\boldsymbol{C}_r \boldsymbol{I}_r, \boldsymbol{C}_g \boldsymbol{I}_g, \boldsymbol{C}_b \boldsymbol{I}_b\}$
Notice that all the vectors in this equation are **unit length vectors**. You need to normalize your vectors to ensure this is true.

The material properties (see the `Material` class) are:

- Ambient reflectance m_a is always visible, regardless of lights in a scene. It multiplies directly with the color $\boldsymbol{c}$ to calculate the ambient color of an object.
- Lambertian reflectance m_l is matte reflection directly related to light falling onto the object from a light source.
- Specular term m_s is the mirror-like reflection of light off an object to the eye.
- Reflection term m_r is the mirror property of the material, which attenuates the bounce ray.
- Metalness m_{sm} controls the color of the specular highlights. $m_{sm} = 0$ means the highlight is the color of the lightsource, $m_{sm} = 1$ means the highlight is the color of the object.
- Specular Exponent (or Phong exponent) m_{sp} characterizes the smoothness (i.e., the sharpness of the highlight spot) of a material, and forms an exponent in the calculation of the specular term.

Let's consider the parts of this equation:

2.1 Ambient Lighting/Shading

Light reflects around a room, illuminating objects uniformly from all sides. This ambient light mixes diffusely, component by component, with the inherent color of the object.

2.2 Lambertian Lighting/Shading

We assume that surfaces are **Lambertian**, thus they obey *Lambert's Cosine Law*:
$\Longrightarrow$ absorbed and re-emitted light energy $\boldsymbol{c} \propto \cos(\theta_{incidence})$, in other words $\boldsymbol{c} \propto \hat{\mathbf{n}} \cdot \hat{\mathbf{I}}$

This states that the color of a point on a surface is independent of the viewer, and depends only on the angle between the surface normal and the incidence vector (the direction from which light falls on the point). We want the actual color to depend on both the color of the light source $\boldsymbol{I} = (r, g, b)$ and the material's color and lambertian reflectance, specified by $m_l \boldsymbol{C}$:

$$\boldsymbol{\rho}_{\text{lambert}} = m_l \boldsymbol{C} \boldsymbol{I} \ \max(\hat{\mathbf{n}} \cdot \hat{\mathbf{I}}, 0) \tag{1}$$

Where ρ_{lambert} is an (r,g,b) intensity value.
$\hat{\mathbf{I}}$ is just the vector defining the light's direction (pointing *at* the light).
$\hat{\mathbf{n}}$ needs to be calculated for the surface itself. Luckily this is easy for spheres, since the normal vector simply points away from the center (see Section 2.4.1).

2.3 Specular Lighting/Shading

The Phong illumination model states that there may be a bright highlight caused by the light source on the surface. This effect depends on where the viewer is. The effect is the strongest when the viewer vector and reflectance vector are parallel.

$$\boldsymbol{\rho}_{\text{specular}} = m_s \boldsymbol{S} \boldsymbol{I} \max(-\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}, 0)^{m_{sp}} \tag{2}$$

The color, $\boldsymbol{S}$, of this highlight is calculated by linearly interpolating between $\boldsymbol{C}$ and $(1, 1, 1)$ according to m_{sm}, the **metalness**. A metalness of 1 means that the specular component takes the color of the object, and a metalness of 0 means that the specular component takes the color of the infalling light.
m_{sp} is the **smoothness** of the material - it affects how small and concentrated the specular highlight is.
$\hat{\mathbf{u}}$ is calculated along the (backwards) viewing ray, from the viewer to the surface point.
$\hat{\mathbf{r}}$, the reflectance vector, is calculated using $\hat{\mathbf{I}}$ and $\hat{\mathbf{n}}$:

$$\hat{\mathbf{r}} = -\hat{\mathbf{I}} + 2(\hat{\mathbf{I}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{3}$$

2.4 Shadows

When doing lighting calculations, we must check if the light is occluded by some intervening object. To do this, we set up a "shadow ray" from the surface point to the light, and check if any objects intersect it. Note that we can re-use our function for intersecting a ray with the scene (`World::intersect`)! For direction lights, any positive hit time of the ray corresponds to a shadow. For point lights, only hit times corresponding to intersections between the surface point and the light position are relevant. (How can you set up the ray so the "shadow" range of hit times corresponds to (0, 1]?)

When a point is in shadow, only the ambient component of the shading formula should be used. The diffuse and specular components are negated by the shadow.

3 Tracing rays

Raytracing models the rendering process as shown in this illustration:

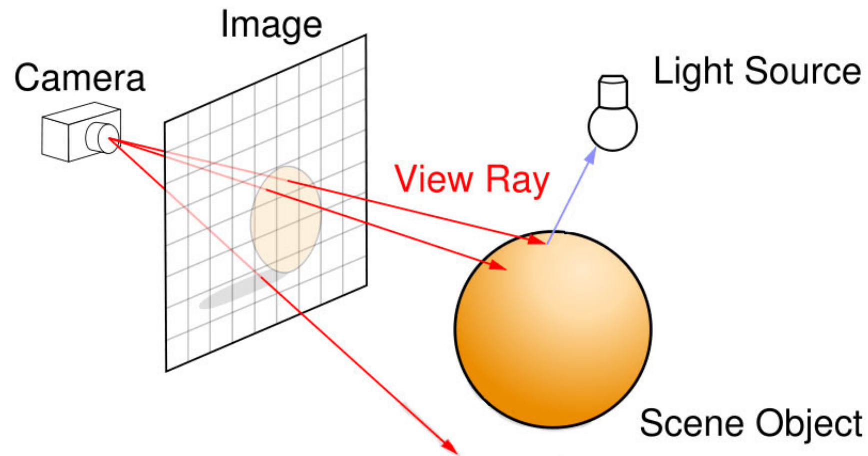

Figure 2: Raytracing

To be able to do this, you need the following components:

3.1 Sampling Viewport

Our viewport (see the `View` class) is a rectangle defined by its 4 coordinates in space, $\vec{\mathbf{LL}}$, $\vec{\mathbf{LR}}$, $\vec{\mathbf{UL}}$, $\vec{\mathbf{UR}}$. We use bilinear interpolation to find a specific point on this rectangle, by varying u and v from 0 to 1 in appropriately sized steps for each pixel. (You don't have to touch this part, it's already coded up for you.)

$$\vec{\boldsymbol{P}}(u,v) = (1-u)\left[(1-v)\vec{\mathbf{LL}} + (v)\vec{\mathbf{UL}}\right] + (u)\left[(1-v)\vec{\mathbf{LR}} + (v)\vec{\mathbf{UR}}\right] \tag{4}$$

3.2 Ray construction

Rays are completely defined by their starting position, $\vec{\boldsymbol{e}}$, and their direction, $\vec{\boldsymbol{d}}$, both of which are vectors. The direction vector can be represented as the difference between a start and an end vector, giving us the familiar linear interpolation formula, and is very useful for constructing rays from the camera to the image:

$$\vec{\boldsymbol{r}}(t) \;=\; \vec{\boldsymbol{e}} + t\vec{\boldsymbol{d}} \;=\; \vec{\boldsymbol{e}} + t(\vec{\boldsymbol{p}} - \vec{\boldsymbol{e}}) \tag{5}$$

Remember that the direction vector need not be a unit vector in our code! The hit times computed by intersection functions are given as multiples of the length of the direction vector. This feature is useful when processing shadow rays – by setting the length appropriately, hit times in (0, 1] correspond to shadows.

3.3 Intersection Tests

We want to find the intersection between a sphere of radius r at position $\vec{\boldsymbol{c}}$ and a ray from position $\vec{\boldsymbol{e}}$ in direction $\vec{\boldsymbol{d}}$. The sphere can be represented as an **Implicit Surface** of the form $f(\vec{\boldsymbol{p}}(t)) = 0$. Finding the parameter t value at which an intersection occurs means solving that equation. From Shirley, page 77, we find the following formula for t, where $\vec{\boldsymbol{d}}$ and $\vec{\boldsymbol{e}}$ describes the ray and $\vec{\boldsymbol{c}}$ and R described the sphere:

$$t = \left(\frac{1}{\vec{\boldsymbol{d}} \cdot \vec{\boldsymbol{d}}}\right) \left[(-\vec{\boldsymbol{d}}) \cdot (\vec{\boldsymbol{e}} - \vec{\boldsymbol{c}}) \pm \sqrt{(\vec{\boldsymbol{d}} \cdot (\vec{\boldsymbol{e}} - \vec{\boldsymbol{c}}))^2 - (\vec{\boldsymbol{d}} \cdot \vec{\boldsymbol{d}})((\vec{\boldsymbol{e}} - \vec{\boldsymbol{c}}) \cdot (\vec{\boldsymbol{e}} - \vec{\boldsymbol{c}}) - R^2)}\right] \tag{6}$$

Remember to check both the roots! One might be positive and the other negative.

You can implement this directly using operators on our matrix library's objects, but you want to check for the value of the discriminant (the contents of the square root) before you complete the calculation. If it is negative, the square root will be imaginary, and no intersection occurred.

3.3.1 Normals to Spheres

A sphere's outward normal, given its center and a point on its surface, is trivial to find. The normal is the vector from the center to the point on the surface, normalized:

$$\hat{\mathbf{n}} \ = \ \text{Normalize}(\vec{\boldsymbol{p}} - \vec{\boldsymbol{c}}) \ = \ (\vec{\boldsymbol{p}} - \vec{\boldsymbol{c}})/R \tag{7}$$

4 Design Ideas

The following documentation will help you in this project:

- Chapters 3 and 14 in Shirley's textbook (+ 2 and 6 for math background).
- Sid's raytracing guide http://fuzzyphoton.tripod.com
- Raytracer Design http://inst.eecs.berkeley.edu/~cs184/sp09/resources/raytracing.htm
- Intersection Tests http://www.realtimerendering.com/intersections.html

4.1 Overall Raytracer Design

In the framework we supply the start and the end phases of the raytracer. We supply something that generates points on the viewport from which you can construct rays, and we supply the frame aggregator to which color values are written. It is up to you to write the missing parts in between. Let's consider a reasonable design for a raytracer:

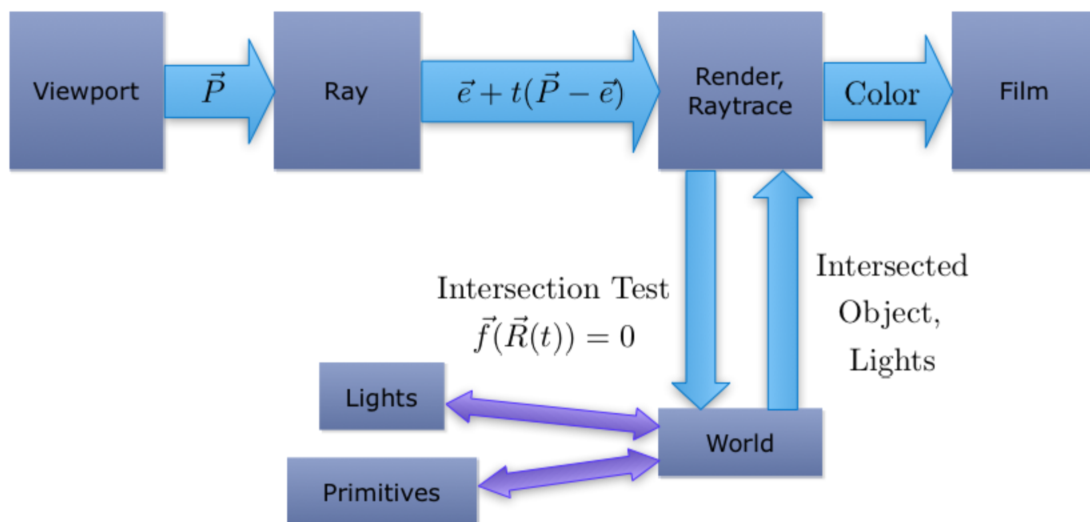

Figure 3: Suggested design

Our raytracer consists of the following:

- **Sampler**:
  - Generates points in world space by using bilinear interpolation across the viewport.
  - The viewport is defined by 4 corners in your scene (aka 4 points in world space), known as LL, LR, UL, UR.
- **Raytrace method**:
  - Rather than making a class for this, we write a `traceRay` method responsible for taking a ray and returning a color.

  - This method will check for intersections of the ray with objects in the scene.
  - This method will calculate shading for objects which were intersected, taking shadows into account.

- **Scene and Geometry**:
  - The scene will store all the geometry in the world, and expose a method to do intersection tests on them.
  - For now, we store geometry (so-called primitives) in a vector in this object.
  - Support the `intersect(ray)` method on a primitive.
  - Geometry / Spheres:
    * Spheres are characterized by a center and a radius. We assume the center is the origin (since arbitrary translations can be applied to the sphere), and store just the radius.

- **Render method:** Since something needs to wire this process together, we have a `renderWithRaytracing` method that gets points from the sampler, traces these points, and saves the pixels to the `Frame` class.

IMPORTANT: We have supplied classes for Color, Ray, Sampled Point and Material in core/Types.hpp, please use these classes to avoid reinventing the wheel. They define many useful operators.

TIP: We suggest that you work first to render primitives without doing any shading or bouncing - just return white if you intersect something, else return black. Next, implement Phong shading. Once this is working, add recursive bounce rays to the raytracer.

5 Results

We will supply you with 5 scene files. You need to render each of these 5 scenes, and they should correspond with the results we show on the assignment page.