Interactive Real-Time Raycasting

CS184 AS4

Due 2009-02-26 11:00pm

We start our exploration of "Rendering"- the process of converting a high-level object-based description into a graphical image for display. We have talked about drawing 2D objects using a process known as **Rasterization**. We will now explore the **Raytracing** approach to rendering. Through the next three projects (AS4 through AS6) you will build an interactive raytracer that will run in semi-realtime.

Figure 1: Ray-traced image

0.1 Required Reading

Please read section 10 through 10.4 and 10.6 of Shirley's "Fundamentals of Computer Graphics" for this assignment, and read the complete Chapter 9 and 10 for the raytracing assignments in general.

1 Shading - Phong Reflectance Model

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

$$\begin{aligned}\boldsymbol{\rho} &= k_a \boldsymbol{CA} + \sum_{Lights} k_d \boldsymbol{CI} \max(\hat{\mathbf{I}} \cdot \hat{\mathbf{n}}, 0) \; + \; k_s \boldsymbol{SI} \max(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^{k_{sp}} \\ \boldsymbol{S} &= (k_{sm})\boldsymbol{C} + (1 - k_{sm})\boldsymbol{(1, 1, 1)}\end{aligned}$$

$\boldsymbol{I} = (r, g, b)$ is the intensity of the infalling light.
$\boldsymbol{A} = (r, g, b)$ is the intensity of the ambient lightsource in the scene.
$\boldsymbol{C} = (r, g, b)$ is the color of the object.
$\boldsymbol{S}$ is the linearly interpolated specular highlight color color according to k_{sm}
$\hat{\mathbf{I}}$ is the incidence vector, supplying the angle of incoming light.
$\hat{\mathbf{n}}$ is the surface normal vector.
$\hat{\mathbf{v}}$ is the vector to the viewer.
$\hat{\mathbf{r}}$ is the reflectance vector, supplying the angle of reflected light.
k_a, k_d and k_s are the ambient, diffuse and specular surface reflection properties 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. Let's consider the parts of this equation:

1.0.1 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.

1.0.2 Diffuse Lighting/Shading

We assume that surfaces are **Lambertian**, thus they obey *Lambert's Cosine Law*:
$\Longrightarrow$ absorbed and re-emmitted light energy $\propto \cos(\theta_{incidence})$, 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 diffuse reflectance, specified by $k_d\boldsymbol{C}$:

$$\boldsymbol{\rho}_{\text{diffuse}} = k_d \boldsymbol{CI} \; \max(\hat{\mathbf{n}} \cdot \hat{\mathbf{I}}, 0) \tag{1}$$

Where ρ_{diffuse} is an (r,g,b) intensity value.
$\hat{\mathbf{I}}$ is just the vector defining the light's direction.
$\hat{\mathbf{n}}$ 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).

1.0.3 Specular Lighting/Shading

The Phong illumination model states that there may be a bright, mirror-like reflection of 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.

$$\rho_{\text{specular}} = k_s \boldsymbol{SI} \max(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^{k_{sp}} \tag{2}$$

The color, $\boldsymbol{S}$, of this highlight is calculated by linearly interpolating between $\boldsymbol{C}$ and $(1,1,1)$ according to k_{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.
k_{sp} is the **smoothness** of the material - it affects how small and concentrated the specular highlight is.
$\hat{\mathbf{v}}$ is calculated between the point on the surface and the viewer position.
$\hat{\mathbf{r}}$, 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 Raycasting

Raycasting models the rendering process as follows:

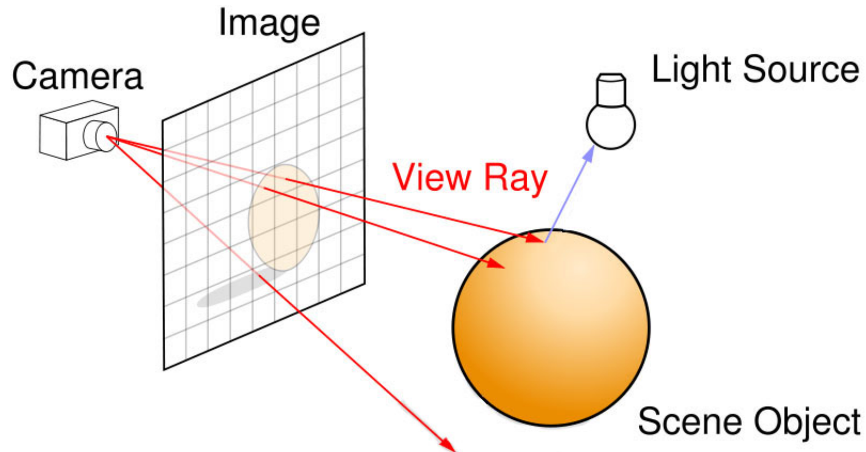

Figure 2: Raycasting

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

2.1 Sampling Viewport

Our viewport 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.

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

2.2 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:

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

2.3 Intersection Tests

We want to find the intersection between a sphere of radius r at position $\vec{c}$ and a ray from position $\vec{e}$ in direction $\vec{d}$. The sphere can be represented as an **Implicit Surface** of the form $f(\vec{p}(t)) = 0$. Finding the parameter t value at which an intersection occurs means solving that equation. From Shirley, page 206, we find the following formula for t, where $\vec{d}$ and $\vec{e}$ describes the ray and $\vec{c}$ and R described the sphere:

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

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.

2.4 Shading Calculations

See section 1 for a detailed discussion on calculating shading.

2.4.1 Normals to Spheres

A sphere's 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:

$$\hat{\mathbf{n}} \;=\; \mathrm{Norm}(\vec{p}-\vec{c}) \;=\; (\vec{p}-\vec{c})/R \tag{7}$$

3 AS4 Minimum Specifications

Write a program called raycaster. It will not take in any input files, and the scene will be hard-coded in your int main function. The raytracer will accomplish at least the following:

1. On load, render the scene described further down with three visible spheres and three light sources, illuminating these spheres using the raycasting method.
2. On pressing "s", save the current frame to disk as a BMP file with the prefix "raycaster"
3. Raycasting:
   - Render a scene by casting, for each pixel on the viewport, a single 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.
4. Shading:
   - Calculate color according to the Phong reflection model.
   - Take each light into account by summing the shading calculation over all lights, using the given direction vector to calculate the necessary angles.
5. Directional Lights:
   - Model lights as emitting an $\boldsymbol{c} = (r, g, b)$ value where each component is in the range of $(0.0, 1.0)$.
   - Model lights as light rays (photons) moving in a direction $\vec{\boldsymbol{d}} = (x, y, z)$, allowing you to calculate the angles needed for shading.
6. Primitives and Material Properties:
   - Support spheres defined by a position $\vec{\boldsymbol{P}}$ and a radius r.
   - Give each sphere 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:
     - Ambient reflectance k_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.
     - Diffuse reflectance k_d is matte reflection directly related to light falling onto the object from a light source. It multiplies with the color $\boldsymbol{c}$ and is scaled by the angle of incoming light to calculate the diffuse color of an object.
     - Specular term k_s is the mirror-like reflection of light off an object to the eye. It multiplies with the color $\boldsymbol{c}$ and is scaled by the angle of incoming light to viewer location to calculate the specular highlight of an object.
     - Metalness k_{sm} controls the color of the specular highlights. $ks_m = 0$ means the highlight is the color of the lightsource, $ks_m = 1$ means the highlight is the color of the object.
     - Specular Exponent (or Phong exponent) k_{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.

   See the Shading section for more details on these terms.
7. Interactivity:
   - Holding down the right mouse button and dragging along the x axis adjusts the metalness k_{sm} of sphere 1.
   - Holding down the right mouse button and dragging along the y axis adjusts the smoothness k_{sp} of sphere 1.

4 AS4 Scene Description

Your scene will consist of an eye location, a viewport location (defined by 4 corners), three spheres and three different light sources. Their properties are as follows:

- **Screen Size:** 512px by 512px
- **Eye:** located at $\vec{e} = (0.0, 0.0, 0.0)$
- **Viewport:** located at:
  - Lower Left: $\vec{\mathbf{LL}} = (-1.0, -1.0, -3.0)$
  - Upper Left: $\vec{\mathbf{UL}} = (-1.0, 1.0, -3.0)$
  - Lower Right: $\vec{\mathbf{LR}} = (1.0, -1.0, -3.0)$
  - Upper Right: $\vec{\mathbf{UR}} = (1.0, 1.0, -3.0)$
- **Sphere 1 (metal):**
  - Center $\vec{p} = (-2.5, -1.5, -17.0)$, radius $r = 2.0$
  - Color $\boldsymbol{c} = \{r = 0.4, g = 0.5, b = 0.9\}$
  - Material Properties $\{ka = 0.1, kd = 0.5, ks = 0.5, ks_p = 150, ks_m = 1\}$
- **Sphere 2 (shiny plastic):**
  - Center $\vec{p} = (0.0, 4.0, -25.0)$, radius $r = 2.5$
  - Color $\boldsymbol{c} = \{r = 0.9, g = 0.4, b = 0.5\}$
  - Material Properties $\{ka = 0.4, kd = 0.2, ks = 0.5, ks_p = 20, ks_m = 0\}$
- **Sphere 3 (chalky):**
  - Center $\vec{p} = (1.5, -1.5, -10.0)$, radius $r = 1.0$
  - Color $\boldsymbol{c} = \{r = 0.5, g = 0.9, b = 0.4\}$
  - Material Properties $\{ka = 0.5, kd = 0.5, ks = 0.3, ks_p = 4, ks_m = 0.5\}$
- **Light 1:** Ambient light, color $\boldsymbol{c} = \{r = 0.5, g = 0.2, b = 0.2\}$
- **Light 2:** Directional light throwing light in direction $\vec{d} = (0.5, 0.5, -0.5)$, color $\boldsymbol{c} = \{r = 0.4, g = 0.8, b = 1.2\}$
- **Light 3:** Point light at position $\vec{d} = (0, 0, -14)$, color $\boldsymbol{c} = \{r = 1.39, g = 0.2, b = 0.2\}$

See the figure below for what we expect this to look like.

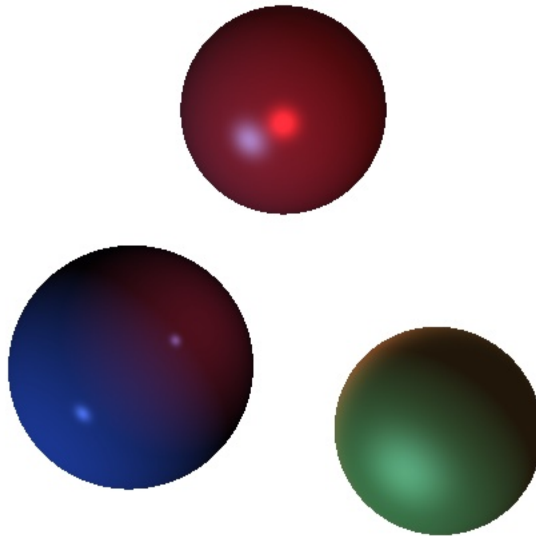

Figure 3: Expected output of given scene with black background removed to save ink. Yours should render with a black background!

5 Design Ideas

From discussion section and lecture you should have a fair idea of how raycasting works. Since as4 is the first assignment on raytracing, this assignment does not ask for a full raytracer, while most of the links concern themselves with a full raytracer. Rest assured that what we do is a strict subset of a full raytracer, and that you will need this information for as5 and as6. Thus, the following documentation will help you for this project:

- Chapter 10 in Shirley's textbook. This chapter is really good and should be your main guide!
- Raytracer Implementation Journal for CS184 Fa07 http://inst.eecs.berkeley.edu/%7Ecs184/sp09/raytrace_journal.
- Raytracer Design http://inst.eecs.berkeley.edu/~cs184/sp09/resources/raytracing.htm
- Intersection Tests http://www.realtimerendering.com/intersections.html
- Fall 2008 notes on shading http://njoubert.com/teaching/cs184_fa08/section/sec03.pdf

5.1 Overall Raytracer Design

In the framework we supply the start and the end phases of the raycaster. We supply something that generates points on the viewport form which you can construct rays, and we supply the Image aggregator to which you write color values for pixels at the end of the raycasting cycle. It is up to you to write the parts in between. Let's consider a reasonable design for a raycaster, similar to what's in the Raytracer Design document:

Thus, we suggest that your Raytracer consists of the following:

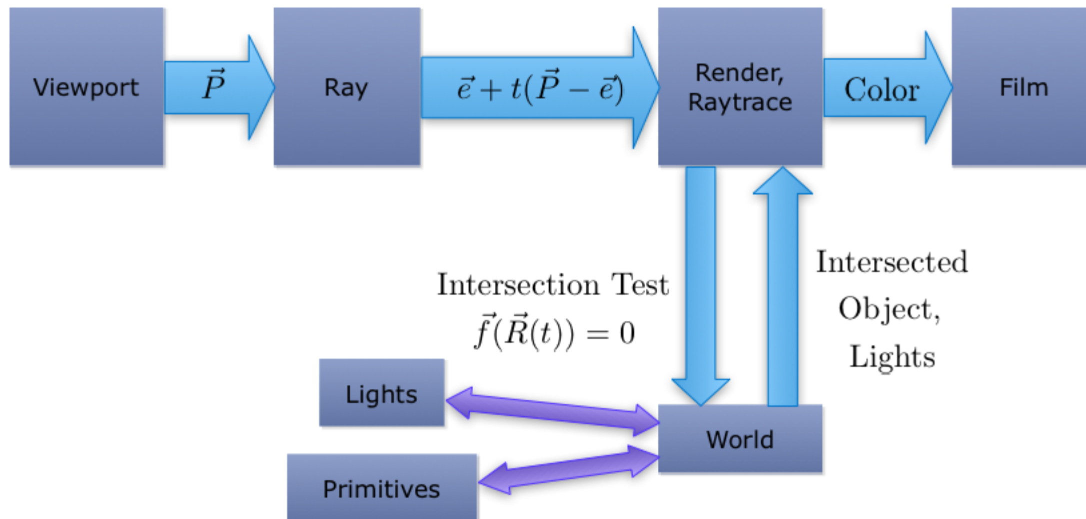

Figure 4: Suggested design

- **Sampler**:
  - Generates points in world space by using Bilinear Interpolation across viewport.
  - Define your viewport as 4 points in your scene (aka 4 points in world space), known as LL, LR, UL, UR.
  - Calling sampler.getPoint() returns the next point to raytrace.
- **Raytrace method**:
  - Rather than making a class for this, write a method inside main 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 object which was intersected.
- **Scene and Gemoetry**:
  - The Scene will store all the geometry in the world, and expose a method to do intersection tests on them.
  - For now, store geometry (so-called primitives) in a vector on this object.
  - Support the primitive = intersect(ray) method
  - Geometry / Spheres:
    - Spheres are characterized by a center and a radius. Store these as a vec4 and a float. (We';; be using homogeneous coordinates.)
    - Write a Primitive class that supports the "intersect(ray)" method. Make "Circle" a subclass.
- **Render method:** Since something needs to wire this process together, write a "Render" method that gets points from the sampler, traces these points, and saves the pixels to the Film class. This is also the main method that we will call to "redraw" the scene if we change something.

IMPORTANT: We have supplied classes for Color, Ray, Sampled Point and Material in algebra3.h, please use these classes to avoid reinventing the wheel. They define many useful operators.