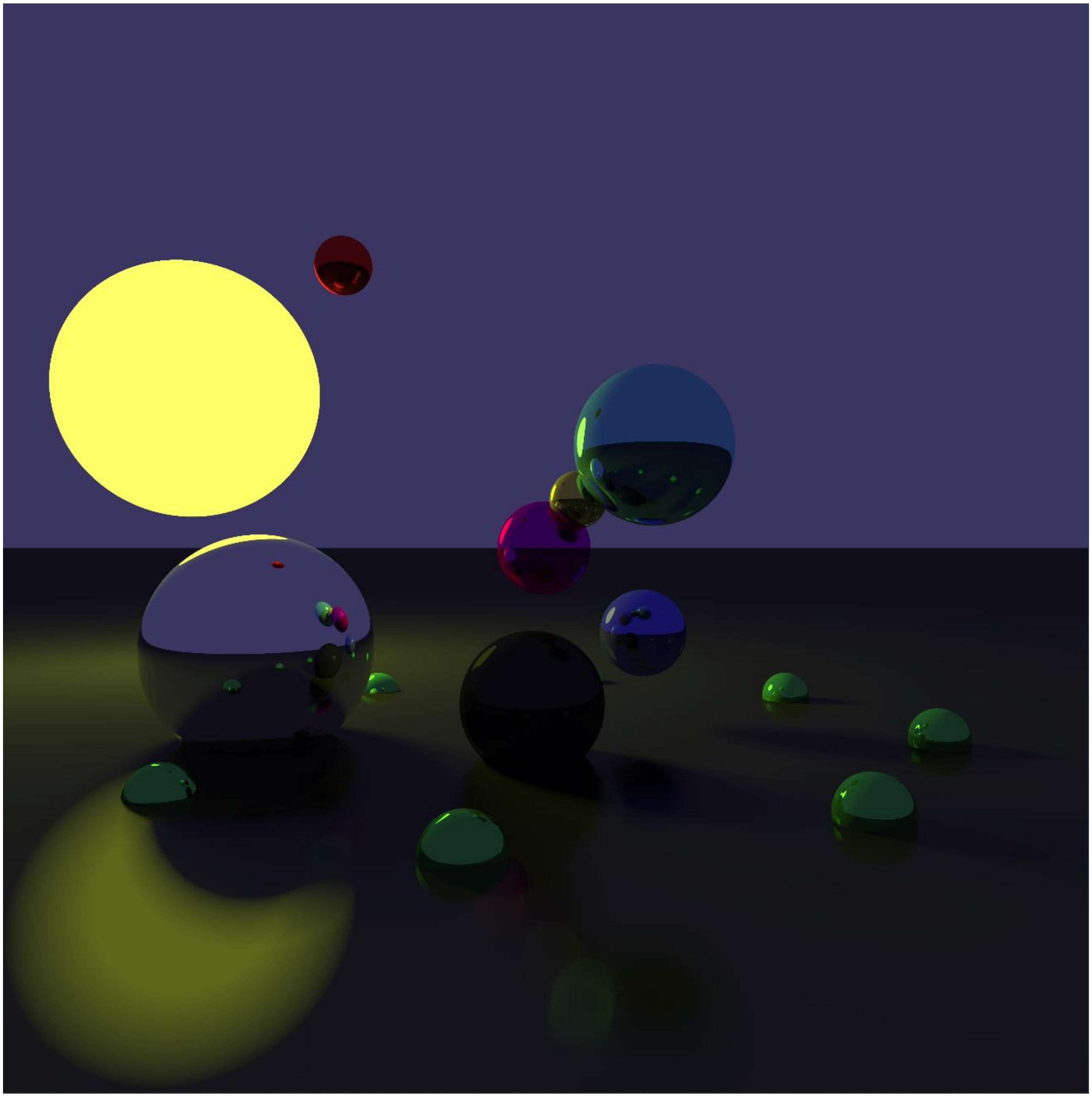
WebGL Path Tracer

CS428 Spring 2020 – Forrest Smith (fcs34)



Outline

Introduction

I've always been interested in game development and have accordingly taught myself as much as I can learn about 3D computer graphics from an artist's perspective, enough to get a game greenlit on the Steam platform. This exposure has taught me that hardware limits real-time realistic light simulation. This forces games to use many rasterization and approximation techniques to "simulate" light. Ray tracing techniques beautifully solve this approximation issue, but use incredibly expensive operations.

Ray tracing essentially mimics the process of photons traveling between a light source and an optical receiver (in this case the camera), the "eye". The operations to calculate these paths these photons take, a ray, is expensive, because when a photon hit a surface, it multiplies and scatters in various directions depending on the surface type. If we only traced one path, we would not have a very realistic, so each time a ray hits a surface it splits into n more paths. This creates an exponential amount of paths to calculate.

Path Tracing is a type of ray tracing that efficiently traces the path from the eye to the light, the reverse of reality. This means we don't have to worry about lost rays from the light that never reach the eye.

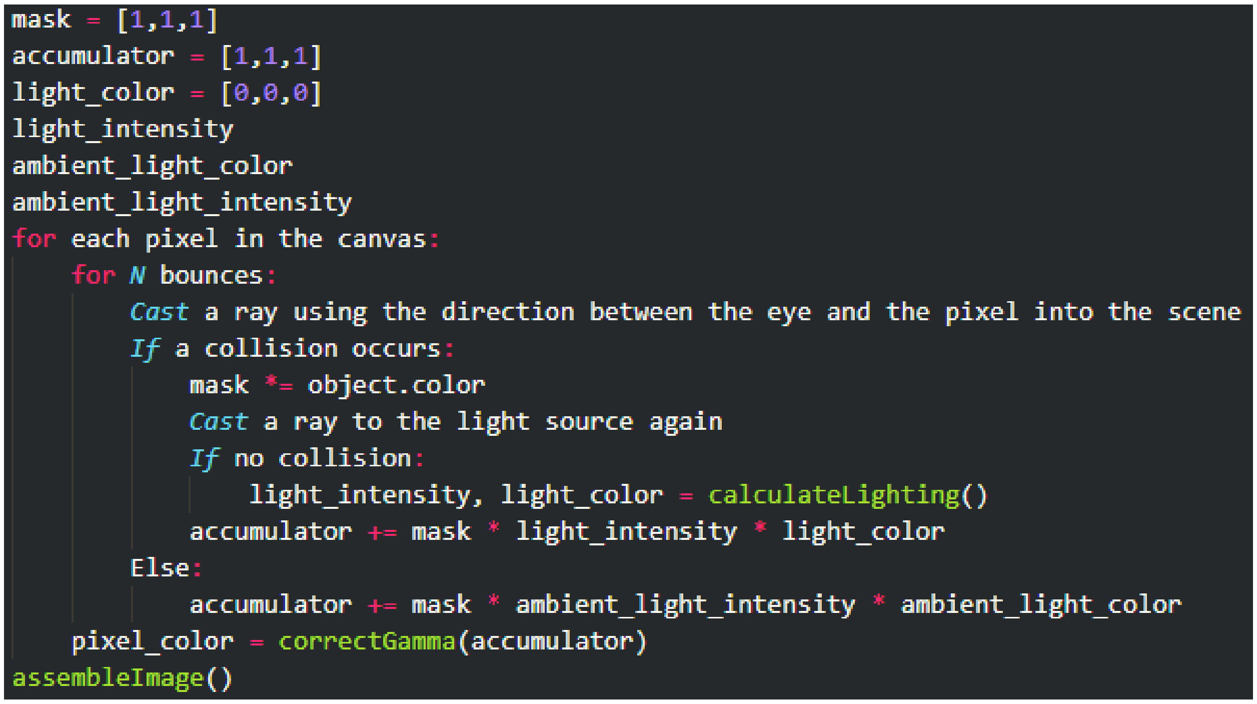
My implementation only accounts for diffuse and reflective surfaces, and soft shadows. The following sections will explain the techniques involved in building this path tracing system.

Basic Algorithm

The image below outlines the basic algorithm for tracing a ray from the eye to the light source, and calculating the resulting fragment. There are some key variables that are referenced in the image that I will explain first:

- Mask – A 3D vector representing color at an intersection
- Accumulator – a 3D vector representing light intensity at an intersection

Essentially, each time we "bounce" our ray, we add and average out our mask and accumulator for the pixel.



```
mask = [1,1,1]
accumulator = [1,1,1]
light_color = [0,0,0]
light_intensity
ambient_light_color
ambient_light_intensity
for each pixel in the canvas:
    for N bounces:
        Cast a ray using the direction between the eye and the pixel into the scene
        If a collision occurs:
            mask *= object.color
            Cast a ray to the light source again
            If no collision:
                light_intensity, light_color = calculateLighting()
            accumulator += mask * light_intensity * light_color
        Else:
            accumulator += mask * ambient_light_intensity * ambient_light_color
    pixel_color = correctGamma(accumulator)
assembleImage()
```

Implementation

Creating the scene

The scene will be constructed and assembled purely in the fragment shader. This allows us to create objects we can easily calculate the intersections with. Utilizing 3D assets would require a more complex solution. So, I decided to create sphere objects and assemble them in the fragment shader. I also created a light object and placed it as well. There is also a floor.

The spheres and the ground will have modifiable roughness values between 0.0 and 1.0 – completely diffuse and completely reflective. This will be very important in how much light is reflected off each surface. This is discussed in the *Bouncing Light* section.

Shooting Rays

Because this is done in the fragment shader, I've opted to make a quad composed of two triangles occupy the entire canvas so that we can render out each ray to the fragments of the triangles.

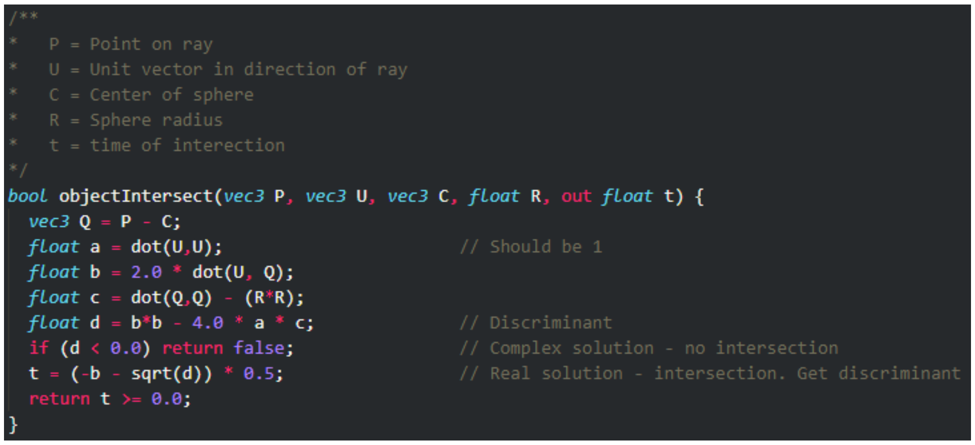
We can calculate the location of the rays by using an inverse projection matrix, to take our eye and projection plane positions and find their according 3D location in the scene.

Line-Sphere Intersection Test

As mentioned before, our scene is comprised of simply spheres (and lights, which are defined identically). We can easily calculate if our ray intersects a sphere by using its position and radius in an implicit function representation. The sphere-intersection tests can be calculated as follows:

Line-Sphere Intersection

I adapted it as follows:



```
/**
*   P = Point on ray
*   U = Unit vector in direction of ray
*   C = Center of sphere
*   R = Sphere radius
*   t = time of interection
*/
bool objectIntersect(vec3 P, vec3 U, vec3 C, float R, out float t) {
  vec3 Q = P - C;
  float a = dot(U,U);                    // Should be 1
  float b = 2.0 * dot(U, Q);
  float c = dot(Q,Q) - (R*R);
  float d = b*b - 4.0 * a * c;           // Discriminant
  if (d < 0.0) return false;             // Complex solution - no intersection
  t = (-b - sqrt(d)) * 0.5;              // Real solution - intersection. Get discriminant
  return t >= 0.0;
}
```

Line-Plane Intersection Test

This was a lot simpler, since our ground plane exists along the x-z plane, so every point of it is at y=0.

I adapted it as follows:

$$t = \frac{groundHeight - rayOrigin.y}{normalizedRayDirection}$$

The interpretation of the possible following results is as follows:

1. **t is 0**: The ray direction is parallel to the ground, and never intersects
2. **t is** ∞: Same as t=0, but is along the plane, and never intersects
3. **t is negative**: The ray is pointing away from the ground and never intersects
4. **t is positive**: The ray points towards the ground and intersects at 1

Intersection Analysis

We can use these equations to find the closest intersection point.

Calculating Color

The mask can easily be updated by multiplying the current mask value by the intersected object's color.

Calculating Light Intensity

In order to calculate the transmission of light, we have to use *shadow rays* which determine if our point is occluded from the light point by another object or part of the current object. A shadow ray is a ray cast from the point of intersection to the light source.

There are three main things to consider when calculating the light intensity:

1. Angle of incidence
2. Apparent size of light source

The angle of incidence of a light determines the area of highlight on a receiving surface. We can calculate the intensity of the this angle of incidence if we normalize the dot product between a normalized shadow ray and the normal of the surface.

Then to get the apparent size of the spherical light source, I used this formula.

Finally, to get the intensity, we just calculate the dot product of scale of the angle of incidence, and the apparent size.

Implementing Soft Shadows

Implementing soft shadows is fairly straightforward. We just have to shoot the shadow ray at a *random* point on the light rather than the center. To do this I made a random RGB noise texture so I could have a normalized 3D value at each pixel, then sampled a random pixel of it, and used that value as an offset for the shadow ray.

Bouncing Light

This part is very straightforward as well. First, a perfectly reflective surface (roughness of 0.0) will reflect a ray perfectly across the normal of the surface. A perfectly diffuse surface (roughness of 1.0) will reflect a ray in any random direction through the surface of a hemisphere above the normal. We can achieve a random direction by using the random noise texture discussed in the previous section. We can express this function as follows if r is our potential ray:

$$r_{new} = mix(r_{reflective}, r_{rough}, roughness)$$

And then we just repeat the process to calculate color, calculate light intensity, and bouncing the ray again.

Random vector/number generation

I mentioned the random offsetting of vectors in the previous two sections. I used this technique to generate a random set of values. Basically, you can generate a texture map with varying depth, and they can then be used as 2D, 3D, etc. vectors. We can send them to the GPU and then sample them every time we want a random offset. If we use a color schema like RGB, we can get a random vec3 associated with every fragment of our canvas.

References

Scratchapixel

https://www.scratchapixel.com/

Explains rasterization, ray & path tracing, and more 3D topics with theory and code examples, including the topics below.

Wikipedia - Path Tracing

https://en.wikipedia.org/wiki/Path_tracing

Another overview of the Monte Carlo Method, Kajiya's rendering equation, etc

Wikipedia - Apparent Size

https://en.wikipedia.org/wiki/Angular_diameter

Overview of apparent size of a sphere, which I apply to the light source

Íñigo Quílez's Gamma Correction

http://iquilezles.org/www/articles/outdoorslighting/outdoorslighting.htm

Explains how to gamma correct an image when using any Ray Tracing algorithm

Improved GLSL rand() Technique

http://byteblacksmith.com/improvements-to-the-canonical-one-liner-glsl-rand-for-opengl-es-2-0/

Monte-Carlo based random number generation