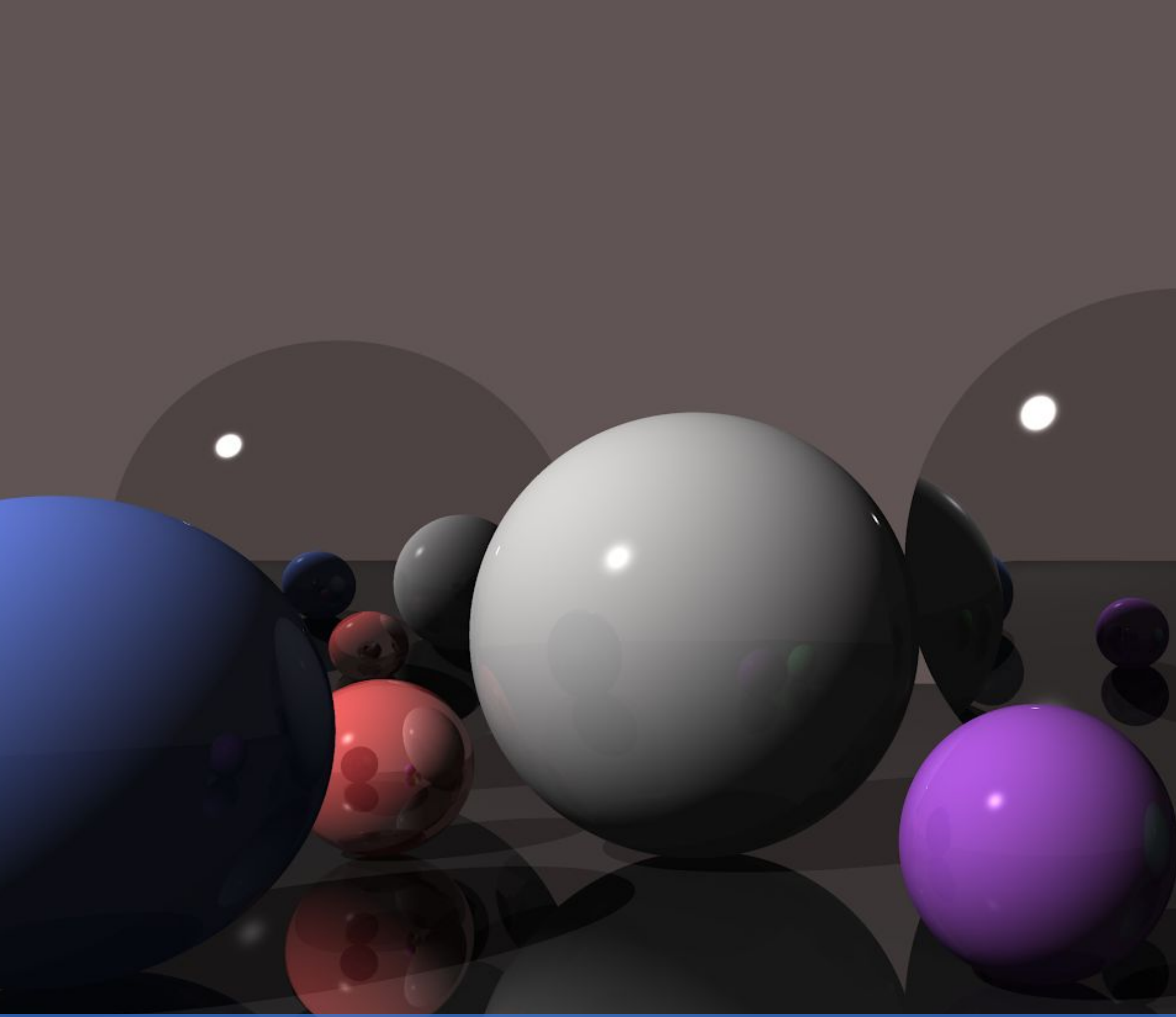


Ray Tracing

Simple Introduction

Dharshan Vishwanatha

RAY TRACING

Simple Introduction

Dharshan Vishwanatha

Contents

Introduction

I wrote this textbook to better understand and retain my knowledge on the basics of ray tracing. But more importantly, help others who have a growing interest in Computer graphics, ray tracing, or lost on what to focus on. Help them get started on building a ray tracer. As ray tracing is a visually appealing project and teaches the fundamentals of programming. It is a very rewarding process, as it's difficult and an impressive end product.

It touches on the programming topics such as OOP Fundamentals, Abstract Classes, Recursion, Multithreading, working with images, and other topics like pointers. But ray tracing is nothing without basic Mathematics fundamentals such as Vectors and how dot products helps us relate two vectors. One thing to mention is that, all the code is on my GitHub: **Ray-Tracing**.

From this book, I hope people either read it, use it as a reference, or even breeze through it. All I want from this book is to inspire others to pursue Computer graphic and ray tracing. And to give the basic background information on how recent advancements in ray-tracing done by NVIDIA with their RTX cards and recent popularity for ray tracing in PS5, Xbox X, and Unreal Engine 5. Without further, we will start with creating our first solid color image.

Chapter 1

Creating an Image

After reading this chapter you should be able to:

- Create PPM images
- Gradient color images

1.1. Image File Format

Since Ray Tracing involves creating images by assigning each pixel value as-pecific color, we can store it in common image formats such as png, jpg, and much more. Although there are libraries such as LodePNG and other ones for different formats to create images. Instead, we will use the NetpbmPPM format to create our images. This is an easy way of creating from a full-color image to black and white images with minimal header and no external libraries. For our case, we will stick to the (0-255) RGB color channel.

1.2. PPM File Structure

The way the ".ppm" file is structured makes it easy to write and read the image file.

```
P3  #Color  channel
<width> <height> <max  color  value>
<R G B> <R G B>  . . .  <R G B>
```

Since we store pixel values linearly and sequentially, we can first write the initial header which tells us what type of color channel and our desired image width and height. Secondly, dump all our pixel values either one-by-one or using a buffer. Note that only P3 needs to have a new line, the rest do not, and each value can be split by space.

1.3. Sample Image Output - Solid Color

Creating a solid red color for a 500x500 image. Since it will be difficult to write out all 250000-pixel values. We can instead do this by code.

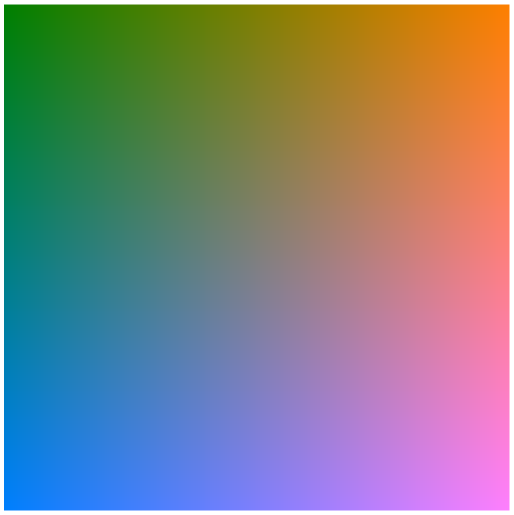
```
int main() {
    uint width = 500;
    uint height = 500;
    const char* fileName = "solidColorImage.ppm";
    ofstream outFile(fileName);          //opens the file to write

    //output P3 500 500 255 as the header
    outFile << "P3\n"
            << width << "␣" << height << "␣255␣";
    for (uint y = 0; y < height; y++) {
        for (uint x = 0; x < width; x++) {
            //output single solid red color
            outFile <<‘‘255 0 0’’<<endl;
        }
    }
    outFile.close();

    return 0;
}
```

1.4. Sample Image Output - Gradient Color

Creating a gradient color 500x500 image. The majority of the code stays the same, but we just change what colors we are outputting to the file.

```
...
//output P3 500 500 255 as the header
outFile << "P3\n"
        << width << " " << height << " 255 ";
for (uint y = 0; y < height; y++) {
    for (uint x = 0; x < width; x++) {
        //output gradient color
        uint red = 255 * x / width;
        uint green = 127;
        uint blue = 255 * y / width;
        outFile<<red<<" "<<green<<" "<<blue<<endl;
    }
}
outFile.close();
...
```

This code will result in an image that looks like:



Chapter 2

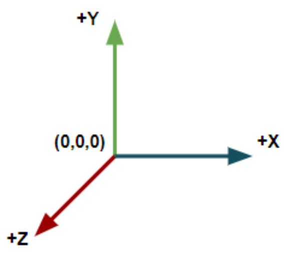
Generating Rays

After reading this chapter you should be able to:

- Getting 3D direction vector, from camera through pixel
- Transforming image/pixel space to screen space

In the real world, where humans live in, light from the sun enters through the atmosphere and bounces around by hitting real-life objects. Eventually, that original light from the sun enters your eye and hits your retina, and you see the world through this way. But in Ray Tracing, we do it backward, since calculating all the bounces is extremely time-consuming and most of them might not hit your eye. And when we start from the eye instead of light, it drastically reduces the computation.

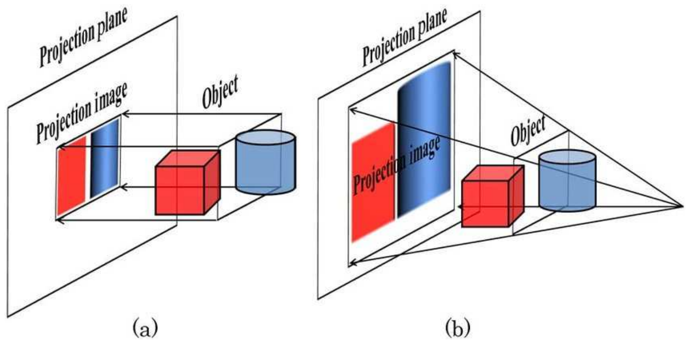
In computer graphics, we tend to use the Right-Hand coordinate system. That is let the y-axis be the virtual axis instead of the z-axis, and now the XZ-plane is the XY-plane. Although everything is up to the programmer, certain things such as culling facing will matter as to which axis we are using.



2.1. Camera and Pixel Grid

Our "eye" is the camera we place in our virtual world by representing as an XYZ-position. Another convection is that the camera is positioned at (0,0,0) and looks down the negative z-axis. And we have a width x height grid of distance d from the camera. This way this will allow us to generate rays from the "eye" through each cell in the grid. This is the fundamentals of Ray Tracing and everything that follows uses these rays to do other calculations.

We have two forms of seeing the world, perspective, and orthographic. The orthographic view is when the camera and the grid are on the same plane (i.e. the distance between them is 0). This makes it easy to generate rays, where each ray starts at each pixel and shoots straight forward. But for perspective is more difficult since our ray origin is at the eye, and the direction needs point towards each pixel.



Credit: **Juan Liu**

2.2. Ray Direction

Our task now is to create a ray that starts at the camera and has a direction towards the pixel coordinates. We call the pixels (image) plane is called the raster plane, and we need to convert this plane into a screen space. Since we define our camera, ray, and other objects later on in the world space, we need to convert our raster plane as well. So that all objects are in the same space. (Need to diff world space vs screen space)

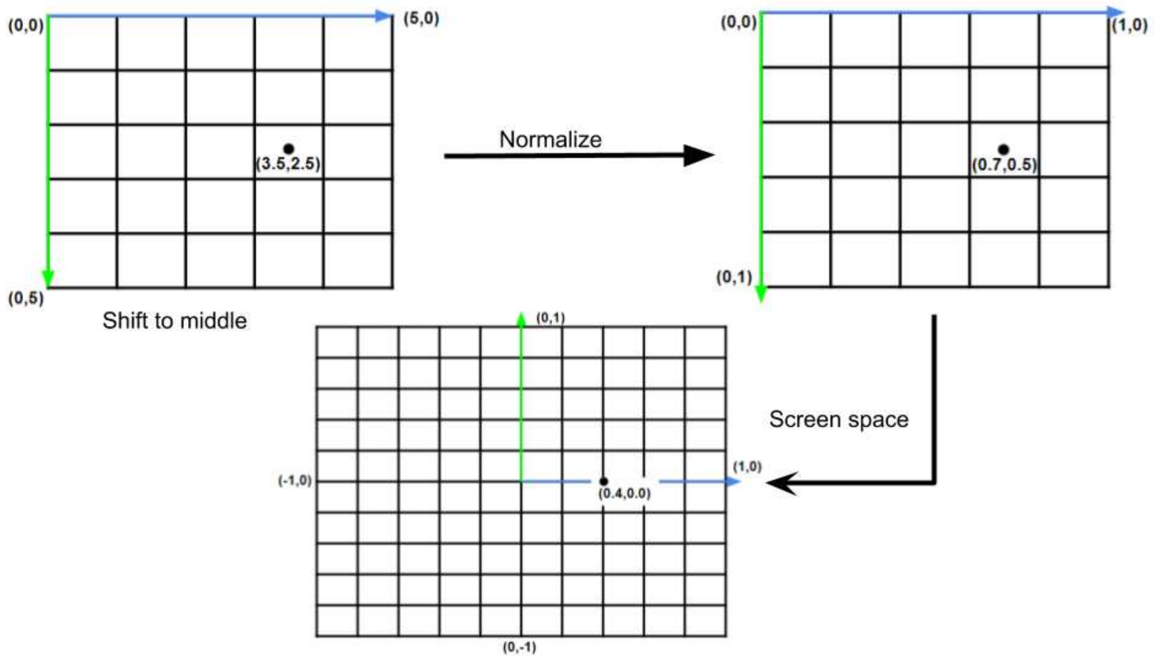
Starting from pixel space, let's say we are trying to create an image of 5x5 pixels, and we want our ray thru the point (x,y), we would move the point (x+0.5,y+0.5) because that would be the middle of the pixel. First, we normalize this point, such that the x and y values are in-between [0,1]. We can normalize by dividing the x component by width, and y by height. Second, we convert these normal points to screen space points which are between [-1,1]. This would make it so that this new point (x',y') would be on the grid in front of the camera, and we can get the ray direction.

The formula for each steps would be:

1. Shift to middle (x,y) $\rightarrow$ (x+0.5,y+0.5)
2. Normalize: (x,y) $\rightarrow$ (x/w,y/h)
3. Screen space: (x,y) $\rightarrow$ (2*x - 1,1-2*y)

For example, consider the pixel (3,2), and we would like to get the color of the pixel. Applying the formula at each step:

1. Shift to middle: (3,2) $\rightarrow$ (1.5,3.5)
2. Normalize: (1.5,3.5) $\rightarrow$ (0.7,0.5)
3. Screen space: (0.7,0.5) $\rightarrow$ (0.4,0.0)



But, we are not done yet. All of this calculation assumes that our image is a square and we didn't take into account the FOV. First, we might not always have a square screen, so we need to take into account for aspect ratio. The aspect ratio = width/height (assuming that width > height). So we just need to multiply the x' by aspect ratio. Second, to identify the cameras FOV, we multiply the x' by tan(alpha/2), where alpha is the camera's FOV angle.

Finally let (x, y) = pixel coordinate, (x', y') = screen space coordinate, w, h = width and height image, $aspectRatio = \frac{w}{h}$, α = FOV. And our screen space coordinate (x', y'):

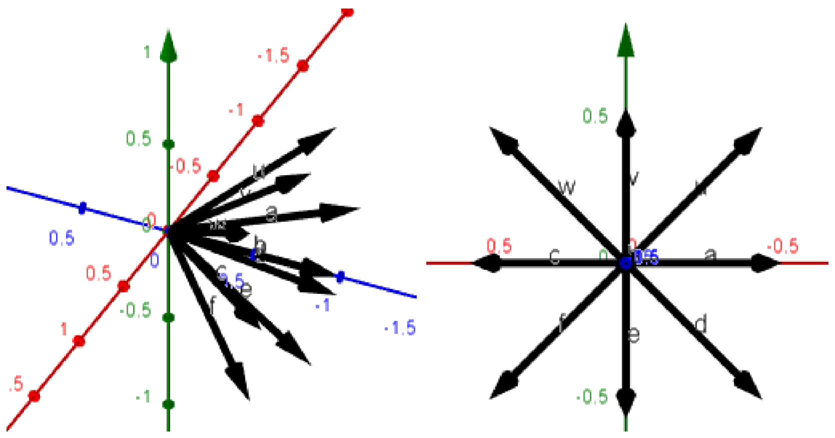
$$x' = (2 \times (\frac{x+0.5}{w}) - 1) \times aspectRatio \times tan(\frac{\alpha}{2})$$
$$y' = (1 - 2 \times (\frac{y+0.5}{h})) \times tan(\frac{\alpha}{2})$$

Since now we have the screen space coordinate (x',y'), we can construct the ray direction by (x',y',-1) - (c_x,c_y,c_z), where (c_x,c_y,c_z) is the camera origin. Since it is a direction, we normalize it.

Translating this to code, we can create a function called *getRayDirection*, which takes in the required parameters: x,y,width,height, and FOV to calculate screen space coordinates. This will be a temporary function since we don't need to copy width, height, and FOV over and over again, and we can just access them from the outside.

```
vec3f getRayDirection(uint x, uint y, uint width,
                      uint height, uint fov) {
    vec3f rayDirection;
    float aspectRatio = (float)width / height;

    //shift to middle, normalize, and screen space
    float xPrime = (2.0f * ((x + 0.5f) / width) - 1) * aspectRatio
                   * tan((fov / 2.0f) * (PI / 180.0f));
    float yPrime = (1 - 2.0f * ((y + 0.5f) / height) *
                   tan((fov / 2.0f) * (PI / 180.0f)));

    //get ray direction towards negative z-axis
    vec3f cameraOrigin(0);
    rayDirection = vec3f(xPrime,yPrime,-1)-cameraOrigin;
    return rayDirection.getNormalized();
}
```

Running this code on all pixels on a 3x3 image and a FOV of 90°. We can compute each ray direction and visualize them on a 3d software by plotting each vector.



Chapter 3

Ray Intersections

After reading this chapter you should be able to:

- C++ inheritance and its uses
- Ray parameterization
- Understand ray-sphere and ray-plane intersection

Having gotten our ray direction for each pixel. Our task now is to trace this ray through our objects and see if this ray intersects any of them. If it doesn't then we just output a background color since we don't intersect anything. If we do, then we ask what we hit, so that we can better understand what the color for pixel should be.

3.1. Generic Objects

First, we build a generic object that all objects inherit from. This makes it easier because we might many different objects like a plane, box, sphere, triangle, and much more. But our ray only cares if it hits an object or not, and what proprieties does it have at that point. For now, all we care about is if he intersects, gets its material propriety, and the normal at the point. Hence, that's all our Object calls has:

Listing 3.1: Object.h

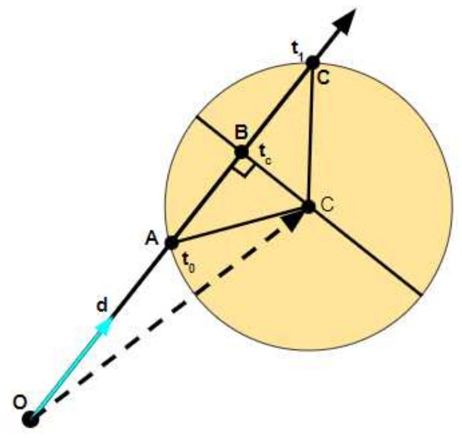
```
class Object {
   public:
    //Check if given ray intersects a given object
    //and if it does return true and modifie the t value
    virtual bool rayIntersect(const vec3f& origin,
                        const vec3f& dir, float& t) const = 0;
    virtual const Material* getMaterial() const = 0;
    virtual vec3f getNormal(const vec3f& hit) const = 0;
};
```

So, if all our object inherits from this. We can call these functions always on that object no matter what type it is. And those objects can write their own definition for these functions. For example, consider the Sphere class. Only the important code is displayed, but the full code can be viewed on the GitHub.

3.2. Sphere Object

A Sphere-ray intersection is one of the easiest forms of the intersection. It starts off defining our sphere is the 3d space, and we only need 2 properties: position and radius to define a sphere. So let (x, y, z) be the position and r the radius. Since we are doing ray intersection, let $\mathbf{r}(t) = \mathbf{O} + \mathbf{D}t$ be the ray parameterization by t, and $\mathbf{O}$ is a origin of the ray, and $\mathbf{D}$ is the direction of the ray. So our goal is to find the t such $r(t)$ is on the surface of the sphere, indicating that the ray intersected the sphere.

First, consider looking at the figure to understand what is happening:



So we want to know when your ray intersects a sphere at points A and B, and what t does it happen. Or it doesn't what values would our t be.

The algorithm for checking ray-sphere intersection:

1. Create a vector $\mathbf{oToC} = \mathbf{C} - \mathbf{O}$. Let $t_c = \mathbf{oToC} \cdot \mathbf{D}$. This value t_c tells us, what scalar value for d gives us the vector which is closet to the center of sphere, which is the point **B** in the figure.
2. Make a right triangle with the points OBC, and if the length of BC is bigger than the radius, then this ray does not intersect and return false.
3. Let α be the length of BC by applying the Pythagorean theorem. Then again to get the length of AB, let this length be $deltaT$.
4. Then $t_0 = t_c - deltaT$ and $t_1 = t_c + deltaT$. This indicates that $\mathbf{r}(t_0) = \mathbf{A}$ and $\mathbf{r}(t_1) = \mathbf{C}$.
5. If both t_0 and t_1 is positive, then ray intersects two points, so use the closest point (t_0). If both t_0 and t_1 is negative, then the ray origin is past the sphere center and radius. So return false.

The C++ code for the sphere-ray intersection is defined $rayIntersect$ which is inherited by Object class, and full code can be seen on the GitHub.

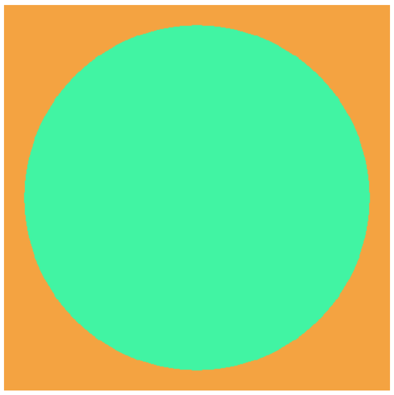
Listing 3.2: Sphere.h

```
bool rayIntersect(const vec3f& origin, const vec3f& dir,
                  float& t) const {
    vec3f oToC = position - origin;
    float t1 = dir.dot(oToC);  //t1 when ray closest to sphere
    float rayToC = oToC.dot(oToC) - t1 * t1;
    if (rayToC > radius) return false;
    float deltaT = sqrt(radius * radius - rayToC);
    t = t1 - deltaT;  //below point
    float abovePoint = t1 + deltaT;
    //check if the origin is inside left of center
    //if it is then t will be zero since will
    //have to travel back to meet below point
    if (t < 0) t = abovePoint;
    //if that the above point is still negative
    //then sphere is behind origin
    if (t < 0) return false;
    return true;
}
```

We can see the sphere intersection method in action by creating the object, the image, and our ray. Using our *getRayDirection* from before, we can test a sphere-ray intersection. If hits, then output a teal color, if not output light orange color. The output of the image is below.

Listing 3.3: main.cpp

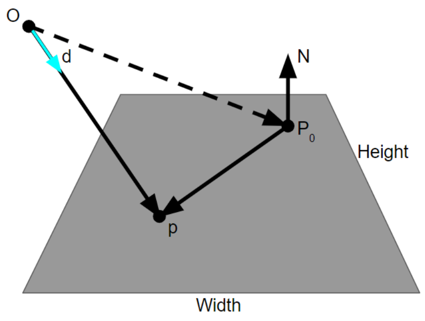
```
vec3f rayOrigin(0);
Sphere sphere(vec3f(0, 0, -1.5), 1);
for (uint y = 0; y < height; y++) {
    for (uint x = 0; x < width; x++) {
        float t;
        vec3f rayDir = getRayDirection(x, y, width, height, FOV);
        if (sphere.rayIntersect(rayOrigin, rayDir, t)) {
            //teal color sphere
            writeColor(out, vec3f(0.258f, 0.960f, 0.643f));
        } else {
            //light orange sphere
            writeColor(out, vec3f(0.960f, 0.643f, 0.258f));
        }
    }
}
```



3.3. Plane Object

Extending from the idea of Sphere class, we also extend the Plane class from the Object class, so that we inherit all the common methods. So to define a plane, we need its center position ($\mathbf{P_0}$), it's normal ($\mathbf{N}$), and width and height. Everything follows from Sphere idea, we need to find t such that $\mathbf{r}(t)$ is on the defined plane.

Again, first, consider the visual representation of Plane-ray intersection:



We know that any vector on the plane and dotted with $\mathbf{N}$ is 0, because they are perpendicular. So if we had a point $\mathbf{P}$ that the ray intersects on the plane, then $(\mathbf{P} - \mathbf{P_0}) \cdot \mathbf{N} = 0$, and using our ray function to the point $\mathbf{P}$, we get $\mathbf{r}(t) = \mathbf{P} = \mathbf{O} + \mathbf{D}t$.

Combining both equations, we get:

$$(\mathbf{O} + \mathbf{D}t - \mathbf{P_0}) \cdot \mathbf{N} = 0$$

$$\mathbf{O} \cdot \mathbf{N} + \mathbf{D}t \cdot \mathbf{N} - \mathbf{P_0} \cdot \mathbf{N} = 0$$

$$\mathbf{D}t \cdot \mathbf{N} + (\mathbf{O} - \mathbf{P_0}) \cdot \mathbf{N} = 0$$

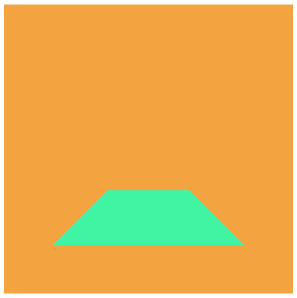
$$\mathbf{D}t \cdot \mathbf{N} = (\mathbf{P_0} - \mathbf{O}) \cdot \mathbf{N}$$

$$t = \frac{(\mathbf{P_0} - \mathbf{O}) \cdot \mathbf{N}}{\mathbf{D} \cdot \mathbf{N}}$$

Listing 3.4: Plane.h

```
bool rayIntersect(const vec3f& origin,
                  const vec3f& dir, float& t) const {
    float denom = dir.dot(N);
    if (fabs(denom) > .0001f) {
        t = ((p0 - origin).dot(N)) / denom;
        //get our point on the plane
        vec3f hitPoint = origin + dir * t;
        vec3f p0ToHit = p0 - hitPoint;
        //check if that point is out of bounds
        if (fabs(p0ToHit[0]) > width ||
            fabs(p0ToHit[2]) > height) return false;
        return t >= 0;
    }
    return false;
}
```

Again, the full code for our Plane object can be found on GitHub. So adding the Plane object at $\mathbf{P_0} = (0, -1, -2.5)$, $\mathbf{N} = (0, 1, 0)$, and width and height of 1. Applying these changes to our main.cpp form our Sphere run, and changing *sphere.rayIntersect*(...) to *plane.rayIntersect*(...) will give use the image of the plane.



Chapter 4

Renderer

After reading this chapter you should be able to:

- Creating a rendering class
- Abstracting functionality
- Building class for extendability

We are at the stage where we can render multiple objects in one image. So we need better automating this process and making it easier for ourselves. Building a rendering class will help us automate the heavy workload of creating objects, managing objects, and creating our image.

Before we begin, we need to look into the future on what need the end product to look like. Consider our main function:

Listing 4.1: main.cpp

```
uint width = 500;
uint height = 500;
uint FOV = 90;

Renderer renderer(width, height, FOV);

renderer.addObject(...);
renderer.addObject(...);

renderer.render();
renderer.output("renderedImage.ppm");
return 0;
```

So, we create a Renderer, add objects to it, call the render function, and output the rendered image to the given file name. This looks much better, looks very readable, and is flexible for future advancements. From that, we can build the Renderer.h file that outlines our Renderer and its functionality (full documentations are on the GitHub). It does not contain the full renderer for ray tracing, just what we have covered so far: if intersect color teal, if not light orange.

Listing 4.2: Renderer.h

```
class Renderer {
   public:
    Renderer(uint width, uint height, uint FOV);
    //adds the object to the list
    void addObject(Object* object);
    //loops thru each pixel and ray intersects each object.
    //And stores it in buffer
    void render();
    //outputs the rendered buffer to ppm file
    void output(const char* fileName);
    ~Renderer();
   private:
    uint index(uint i, uint j);
    vec3f getRay(uint x, uint y);
    //casts a ray through each pixel, calls sceneIntersect,
    //and returns the pixel color
    vec3f castRay(const vec3f& rayOrigin, const vec3f& rayDir);
    //check if given ray intersects any object in the scene
    bool sceneIntersect(const vec3f& rayOrigin, const vec3f& rayDir);
    //clips the value n, between low and high
    float clip(float n, float lower, float upper);

   private:
    uint width;
    uint height;
    uint FOV;
    vec3f cameraOrigin;
    vector<vec3f> buffer;
    list<Object*> objects;
};
```

The definition of each member function is on GitHub. The most important function listed here are: $render$, $output$, $castRay$, $sceneIntersect$. Some of these are not set in store as they will drastically change to handle additional features.

4.1. render()

Listing 4.3: Renderer.h/render

```
for (uint y = 0; y < height; y++) {
    for (uint x = 0; x < width; x++) {
        buffer[index(x, y)] =
            castRay(cameraOrigin, getRay(x, y));
    }
}
```

All it does is assign pixel buffer each value, by calling *getRay* on each pixel, and its color calculated by *castRay*.

4.2. castRay()

Listing 4.4: Renderer.h/castRay

```
if (sceneIntersect(rayOrigin, rayDir)) {
    return vec3f(0.258f, 0.960f, 0.643f);
} else {
    return vec3f(.960f, 0.643f, 0.258f);
}
```

This is our simple *castRay* for now since later on it will handle shadows, reflection, and other features. It checks if our ray intersects any objects in the scene. If so, return teal. If not, return orange.

4.3. sceneIntersect()

Listing 4.5: Renderer.h/sceneIntersect

```
float maxDistance = __FLT_MAX__;
for (const Object* o : objects) {
    float t;
    if (o->rayIntersect(rayOrigin, rayDir, t) && t < maxDistance) {
        maxDistance = t;
    }
}
return maxDistance < RENDER_DISTANCE;
```

For now, our *sceneIntersect* checks for the intersection of each generic object with the given ray, and keeps track of the closes object. And check if our intersection is over the max render distance or not. Later, we will add features that give us additional details on what the ray intersections.

4.4. output()

Listing 4.6: Renderer.h/output

```
for (uint y = 0; y < height; y++) {
    for (uint x = 0; x < width; x++) {
        uint colorIndex = index(x, y);
        for (int i = 0; i < 3; i++) {
            float clipped = 255 * clip(buffer[colorIndex][i],
                                       0.0f, 1.0f);
            u_char pixelValue = (u_char)clipped;
            outputBuffer.push_back(pixelValue);
        }
        if (outputBuffer.size() == outputBuffer.capacity()) {
            out.write((char*)&outputBuffer[0], outputBuffer.size());
            outputBuffer.clear();
        }
    }
}
```

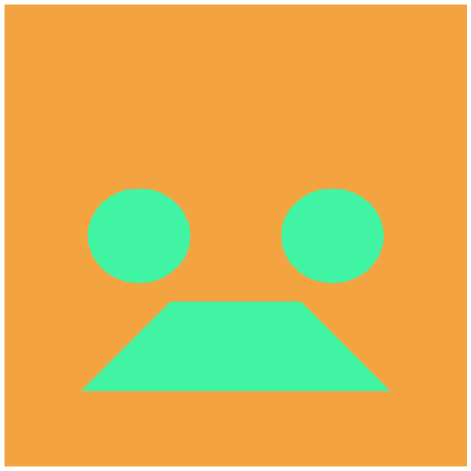
This code will stay the same as this is independent of the ray and just needs the pixel value buffer to output to the file. Full code on GitHub.

The idea is the same when we created a solid color image, but instead, our pixel value color from ray tracing is normalized. So, first, we clip each pixel value between 0 and 1 and convert to 255 range. Instead of outputting one-pixel value by one, which is costly because access to the file and writing one-by-one is an expensive operation. It instead outputs by a temporary buffer of size 300 and refills when the size and capacity are the same.

Finally, since we abstracted the majority of the code in the main.cpp, consider adding 3 objects and its rendered image:

Listing 4.7: main.cpp

```cpp
renderer.addObject(new Sphere(vec3f(-2, 0, -5), 1));
renderer.addObject(new Sphere(vec3f(2, 0, -5), 1));
renderer.addObject(new Plane(vec3f(0,-1,-2.5), vec3f(0,1,0),1,1));
```



Chapter 5

Shadows

After reading this chapter you should be able to:

- How to tell if a pixel is in shadow or not
- Vector operation such as dot product

Shadows are one of the most important features a rendering can have. It is what distinguishes between 2D and 3D. Like in painting, shadows given the illusion of depth, and our rendering is just like a painting. Up until now, we have rendered 3D Spheres and Planes, but they just look like a 2D shape. By adding shadows we can finally render the 3D shapes we defined. To create shadows, we need some sort of light source. So we will start by creating a Light object.

5.1. Light

It is a simple object, it mainly has a vec3 position and a float light intensity variable. And two getters for these two variables. You can add a Material property such that the light can have different colors, but we will stick with white light for now.

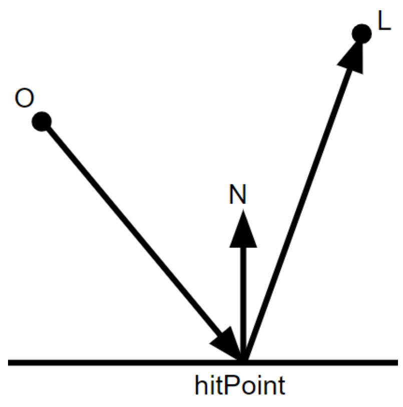
Listing 5.1: Light.h

```
const vec3f* getPosition() const { return &position; }
float getIntensity() const { return intensity; }
vec3f position;
float intensity;
```

Also, as of now. We have built the base for ray tracing, and going further we will modify the given Renderer.h to handle additional features such as shadows.

5.2. Shadow Rays

The idea behind checking if a given point on objects is in shadow or not is tracing another ray from the hit point to all active light sources. This is a common theme in ray tracing if we come across a difficult problem, we just generate more rays. So, we if call the ray from the hit point to the light source, the shadow ray (since we are checking for shadows). Then this point is in shadow if our shadow rays intersect any object towards the light (i.e. light is blocking this point on the object). Consider the figure for a visual explanation:



So, if our pixel ray hits a point, then we create another ray from the hitPoint to the light direction. And if we intersect along the way then, its in shadow.

And if we don't, then the light hits the point. So we modify our *castRay* function to handle this feature because this is where high-level ray and scene intersection takes place.

First, we begin by changing our $sceneIntersect$ function, to handle getting our normal vector N, and where we hit the object. Done it as a reference.

Listing 5.2: Renderer.cpp/sceneInterect

```
bool sceneIntersect(..., vec3f& hitPoint, vec3f& N){
if (o->rayIntersect(rayOrigin, rayDir, t) && t < maxDistance) {
    maxDistance = t;
    hitPoint = rayOrigin + rayDir * t;
    N = o->getNormal(hitPoint).getNormalized();
}
}
```

Next our *castRay* function drastically changes from our simple scene intersection to handle lights and shadows. This will also set the base for reflections, refraction, and other features.

Listing 5.3: Renderer.cpp/castRay

```
vec3f hitPoint, N;
if (!sceneIntersect(rayOrigin, rayDir, hitPoint, N)) {
    //ray intersects nothing, so return orange
    return vec3f(.960f, 0.643f, 0.258f);
}

float diffuseIntensity = 0;
for (Light* l : lights) {
    vec3f pointToLight =
        (*l->getPosition() - hitPoint).getNormalized();
    if (inShadow(l, hitPoint, N, pointToLight)) continue;
    //accumulate all the diffuseIntensity from each light
    diffuseIntensity += getDiffuse(l, N, pointToLight);
}
//return the teal color based on how much light hits it
return vec3f(0.258f, 0.960f, 0.643f) * diffuseIntensity;
```

So, first, we check for the intersection. If so, then we check if our hitPoint is in shadow or not. Loop through each light, get the light direction from hitPoint to Light. If *inShadow* return false, indicates that light hits this point, and increase the diffuseIntensity since it's a brighter there. And the final teal color is based on diffuseIntensity. We see two new functions here, *inShadow* and *getDiffuse*.

Starting with the easy one *getDiffuse*, it simply calculates how parallel the hitPointToLight is with the normal, using the dot product.

Listing 5.4: Renderer.cpp/getDiffuse

```
return l->getIntensity() * max(0.0f, N.dot(light));
```

And finally, *inShadow*, calls *sceneIntersect* starting at the hitPoint origin and checks the distance. But we push the origin a bit forward or backward to avoid self-intersection.

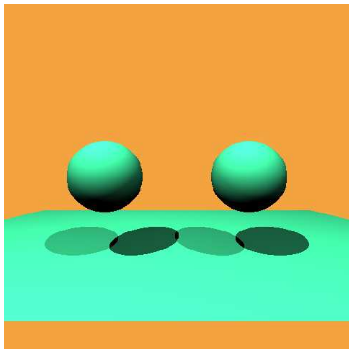
Listing 5.5: Renderer.cpp/inShadow

```
vec3f hitPoint, origin;
float lightDistance = (*l->getPosition()).length();
origin = (dirToLight.dot(hitN) >= 0) ? rayHit + hitN * .001f
           : rayHit - hitN * .001f;
if (sceneIntersect(origin, dirToLight, hitPoint, hitN) &&
    (hitPoint - origin).length() < lightDistance) {
    return true;
}
return false;
```

Finally, after 2 lights in our main.cpp. We end up with this image:

Listing 5.6: main.cpp

```
renderer.addLight(new Light(vec3f(-5, 10, 3), 0.8f));
renderer.addLight(new Light(vec3f(5, 10, 3), 0.5f));
```



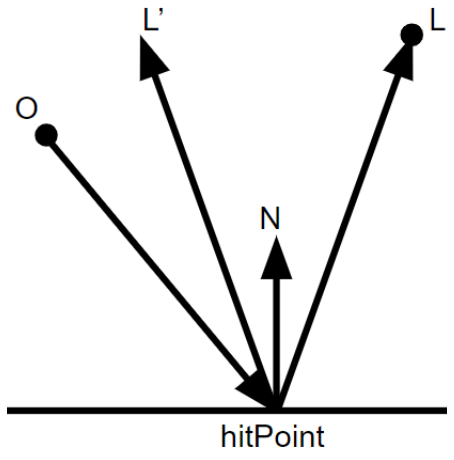
Chapter 6

Specular Highlights

After reading this chapter you should be able to:

- How to reflect a vector across another
- How higher exponent increases specular sharpness

Adding a specular highlight is very similar to shadows. But first, we need to know how and when specular highlight occurs. Again, consider a visual representation first:

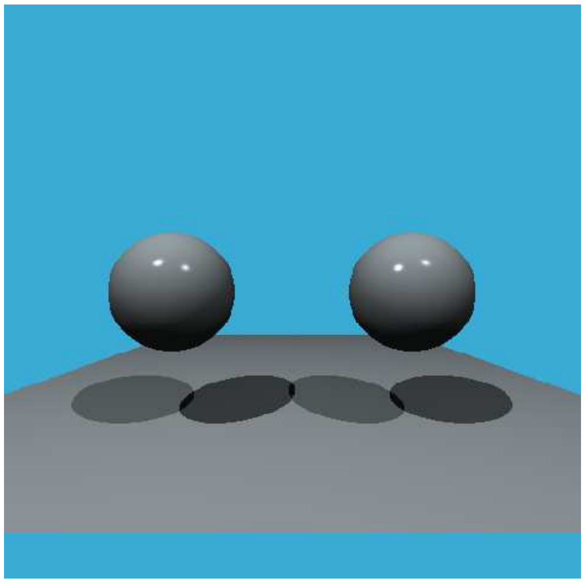


So referring to the figure, we reflect the light vector L across our normal vector. We then dot this reflected vector with our pixel ray (camera ray), to check how much of the incoming light hits our camera pixel. Because dot product tells us how parallel the 2 vectors are. We have to make sure that the pixel ray is reversed for the dot product to give correct results.

Listing 6.1: main.cpp

```
float specularIntensity = 0;
for (Light* l : lights) {
    ...
    vec3f reflectedLight = reflect(pointToLight, N).getNormalized();
    if (inShadow(l, hitPoint, N, pointToLight)) continue;
    ...
    specularIntensity += (l->getIntensity() *
                powf(max(0.0f, reflectedLight.dot(-dir)), 100));
}

return ... + vec3f(1) * specularIntensity;
```

The intensity is again calculated by how parallel the vectors. And we raise it to a power 100. And the higher the power, the sharper the specular highlight. Here we return vec3f(1) scaled by specularIntensity, meaning it is a white light specular highlight. We can modify this vector for different specular color. Finally, we have our specular highlights, which can be seen in the top of the sphere. Changing up colors a bit.



Chapter 7

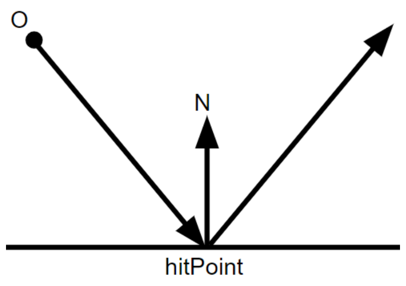
Reflection

After reading this chapter you should be able to:

- Using recursive *castRay* to solve reflection problem

Reflection is at the heart of Ray Tracing, and what differentiates this type of rendering as oppose to Rasterization. This is what pops out and what you will primarily see in Ray Tracing videos from NVIDIA and other videos. But the solution to reflection is as easy as they come in Ray Tracing. To solve the problem of reflection, we just generate more rays. This is why it's not practical to use this type of rendering because all the calculation that is needed just for a single pixel color.

So how exactly do we reflect? When our pixel ray intersects an object, we reflect this vector across the normal, and call *castRay*, from this point. So our hitPoint acts as a camera origin, and we start doing all the calculations over again because we need to get this perspective to get reflections.

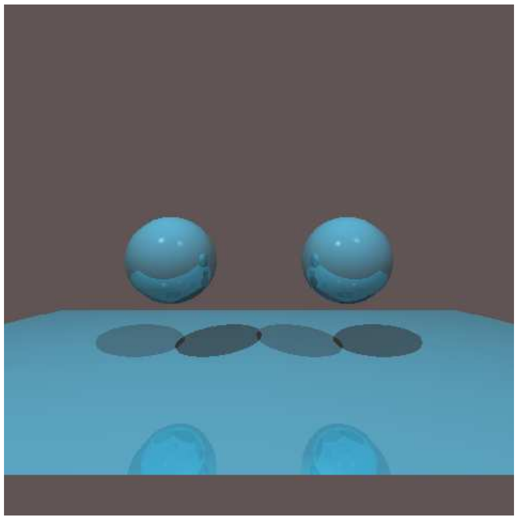


One thing to notice is that, we call *castRay* inside *castRay*. This is a recursive call, and recursion is a great way of solving difficult problems. So, if we do recursion, we need to stop at some point. Hence, we keep a maximum recursion depth value, and if we go over we return. But, maximum recursion depth acts as how many self-reflections we see. And what we return at the end is the background color.

Hence, here is our modification for *castRay*, you see we have other constants when we return, this is the albedo, we will see this in Material.

Listing 7.1: main.cpp

```
if (depth > MAX_DEPTH ||
    !sceneIntersect(rayOrigin, rayDir, hitPoint, N, hitMaterial)) {
    return vec3f(0.384, 0.333, 0.333); //background color
}

vec3f reflectDir = reflect(-dir, N);
vec3f reflectOrigin = getOriginShift(hitPoint,reflectDir,N);
vec3f reflectColor = castRay(reflectOrigin, reflectDir, depth + 1);
...
return vec3f(0.227f, 0.670f, 0.827f) * diffuseIntensity * 0.6f +
    vec3f(1) * specularIntensity *  0.3f +
    reflectColor * 0.5f;
```



Chapter 8

Material

After reading this chapter you should be able to:

- Basic object properties: albedo, color, and specular.

Up until this point, we have only defined two colors, one when we intersect and another for the background. In this chapter, we will use the Material class we created so that we can have more control over the color and look of our objects.

The process is relatively simple. We pass a Material object called hitMaterial in our *sceneInterction* method as a reference. And we modify this value when we hit an object, and equal to *o->getMaterial(). And in our castRay, we use this hitMaterial variable to get the object's properties such as color, specular exponent, and much more. Finally, we will modify what color we return in our *castRay* function. Again, full code can be viewed on GitHub.

Listing 8.1: Renderer.cpp/castRay

```
...
return hitMaterial.diffuse() * diffuseIntensity *
        hitMaterial.albedo()[0] +
        vec3f(1) * specularIntensity * hitMaterial.albedo()[1] +
        reflectColor * hitMaterial.albedo()[2];
```

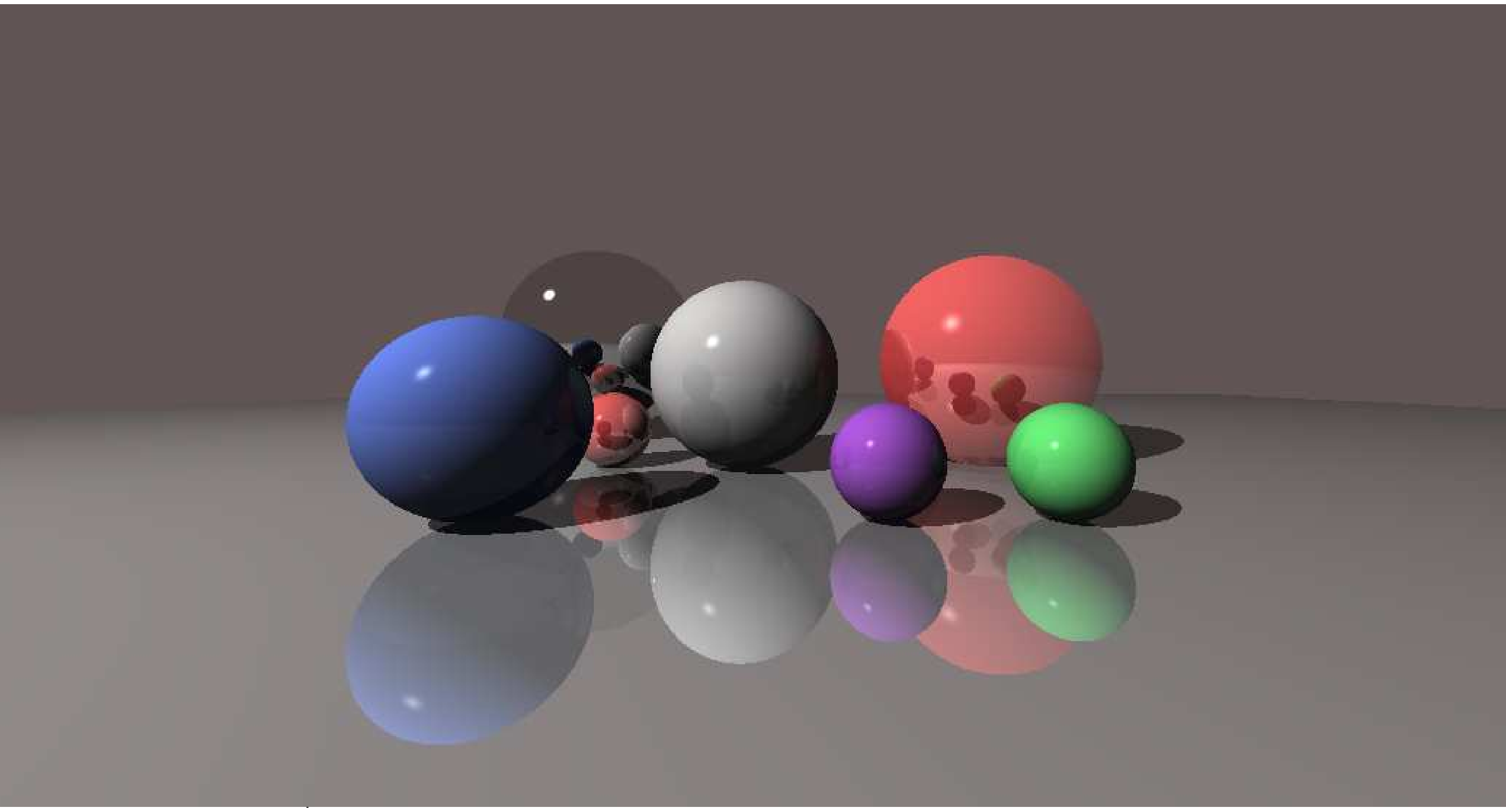
And in our main.cpp, we can define multiple Materials for each object, so that we have different colored objects in our scene.

Finally, from what we all have created and made so far, and if we let our main function be:

Listing 8.2: main.cpp

```
Material redMarble(vec3f(0.6f, 0.3f, 0.5f),
vec3f(0.8, 0.231, 0.231), 200);
Material greyMarble(vec3f(0.6,0.3f, 0.1f),
vec3f(0.850, 0.843, 0.843), 200);
Material purpleMarble(vec3f(0.6, 0.3f, 0.1f),
vec3f(0.686, 0.329, 0.890), 200);
Material greenMarble(vec3f(0.6, 0.3f, 0.1f),
vec3f(0.407, 0.968, 0.482), 200);
Material blueMarble(vec3f(0.6, 0.3f, 0.1f),
vec3f(0.407, 0.529, 0.968), 200);
Material planeMat(vec3f(0.6f, 0.3f, 0.45f),
vec3f(0.509, 0.509, 0.509), 350);
Material mirror(vec3f(0.0f, 10.0f, 0.8f),
vec3f(1.0f), 1425.);

renderer.addObject(new Sphere(vec3f(2,0.15, -6),
1.5, redMarble));
renderer.addObject(new Sphere(vec3f(-0.75, 0, -5.5),
1, greyMarble));
renderer.addObject(new Sphere(vec3f(0.5, -0.6, -3.5),
0.4, purpleMarble));
renderer.addObject(new Sphere(vec3f(1.75, -0.6, -3.5),
0.4, greenMarble));
renderer.addObject(new Sphere(vec3f(-2.3, -0.3, -3.5),
0.7, blueMarble));
renderer.addObject(new Sphere(vec3f(-2.2, -0.6, -5.5),
0.4, redMarble));
renderer.addObject(new Sphere(vec3f(-3.35, 0.5, -8),
2, mirror));
renderer.addObject(new Plane(vec3f(0, -1, -5),
vec3f(0, 1, 0), 20, 20, planeMat));
renderer.addLight(new Light(vec3f(-7, 7, 3), 1.5f));

renderer.render();
renderer.output("renderedImage.ppm");
```

Form that, we create this beautiful image:



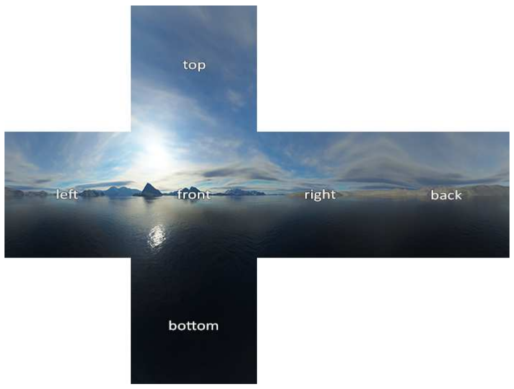
Chapter 9

Cubemaps

After reading this chapter you should be able to:

- Read in images using stb_image.h
- Converting 3D direction into 2D position

Although, the previous render looks pretty impressive (aside from the compression). We will take it a step further and have an interesting background instead of a solid color. This is where cubemaps come into play, as it creates a full 360° background of any image we like. The way cubemaps works is, a photographer takes a 360° image, and converts into 6 2D images. Take for example a cubemap from LearnOpenGL.



You can "fold" this such as you make a cube. Also notice that they are continues from one image to another. This gives the illusion of 360° image.

So when we output a solid background color, we have a variable rayDir, which tells us where our ray is pointing outwards. Hence instead of outputting a solid color, we see where this rayDir is pointing, and pick that color in our 360° image. But we don't have a 360° image, we have 6 images. So we need to instead go into our cubemap images, figure out which out of 6 images, and then return the pixel color from that image. Here is our Cubemap class, important functions. One thing to note is that majority of the implementation is from Wikipedia on **Cube Mapping**.

Listing 9.1: Cubemap.h

```
Cubemap(const vector<const char*>& faces);
vec3f getBackground(const vec3f& direction);
void vectorToUV(float x, float y, float z,
                     int* index, float* u, float* v);
```

Our constructor for Cubemap, reads in all the images from the faces path using stb_image.h, and storing as struct of unsigned char pointer, width, height, and channels.

Listing 9.2: Cubemap.h/Cubemap

```
int width, height, nChannels;
for (const char* pathi : faces) {
    u_char* data = stbi_load(pathi, &width, &height, &nChannels, 3);
    cubemaps.push_back({data, width, height, nChannels});
}
```

Our *getBackground* takes in the rayDir and outputs the color value that the rayDir points to in that 360° image (cubemap). It mainly uses *vectorToUV* function to get the u,v and faceIndex. And uses these variables to the movie to the correct image and pixel coordinate to get the value.

Listing 9.3: Cubemap.h/Cubemap

```
float u, v;
int faceIndex;
vectorToUV(direction[0], -direction[1], direction[2],
           &faceIndex, &u, &v);
```

```
float RGB[3];
uint i = uint(u * cubemaps[faceIndex].width);
uint j = uint(v * cubemaps[faceIndex].height);
uint dataIndex = index(i, j, cubemaps[faceIndex].width);
for (int i = 0; i < 3; i++)
    RGB[i] = (float)cubemaps[faceIndex].data[dataIndex * 3 + i];
return vec3f(RGB[0], RGB[1], RGB[2]) / 255.0f;
```

Next, the *vectorToUV* function is from the Wikipedia page. Finally, to get the cubemap working in our Renderer, we just have to add the Cubemap object and initialize the face paths in main. So our *castRay* changes a bit.

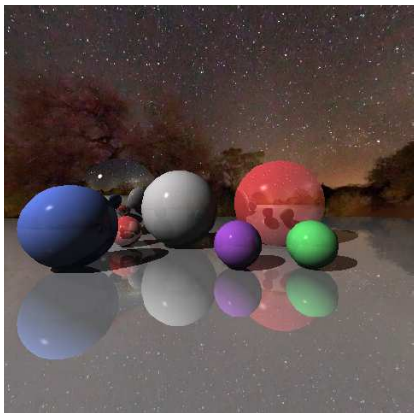
Listing 9.4: Renderer.h/castRay

```
if (depth > MAX_DEPTH ||
    !sceneIntersect(rayOrigin, rayDir, hitPoint, N, hitMaterial)) {
    return useCubemap ? cubemap.getBackground(rayDir) : background;
}
```

So, if we are using cubemap, then we access cubemap's color values, if not we output a default/user-defined solid background color. And in main we define like so, or where ever your Skybox images are located:

Listing 9.5: Renderer.h/castRay

```
vector<const char*> facesPath = {"Skybox//px.png", "Skybox//nx.png",
                                 "Skybox//ny.png", "Skybox//py.png",
                                 "Skybox//pz.png","Skybox//nz.png"};
Renderer renderer(width, height, FOV, facesPath);
```



So instead of having a boring solid color, we have stars and trees now.

Chapter 10

Multithreading Rays

After reading this chapter you should be able to:

- C++ Multithreaded when and how
- Image multithreading

Although we have a working Renderer, which can render any given width and height. But, if we were to render 4k or 8k resolution, we would have to wait about 16 secs for 4k and 70 secs for 8k. This is unreasonable since we are just rendering one image. We can improve this time by implementing multithreading.

So when is a good idea to use multithreading? When we have an independent calculation. Take for example calculating the Fibonacci sequence. This would not be possible with multithreading because the current value relies on the calculation of the previous two. So it's dependent on previous values.

We can use multithreading for our case because each pixel calculation is independent of any other pixels. So in theory, we can assign each pixel its thread, and all of the pixel colors will be calculated at the same time. But the overhead of creating each thread takes resources. And in the end, our time will be much worse.

To implement an efficient multithreaded Renderer, we will consider splitting our image into 4 equal quadrants recursively.

Let *splitCount*, be the number of times we split each image (or subimage). To describe the splits lets go at each split starting from 0.

1. *splitCount* $= 0$, then it is just a single threaded run.
2. *splitCount* $= 1$, then our image becomes a grid of 2x2. Like a xy-graph which has 4 quadrants.
3. *splitCount* $= 2$, then we have 4x4 grid, where each quadrants is 2x2 grid.

And so on. So for each cell in our grid (based on *splitCount*) runs on its thread. This is valid multithreading since each cell is independent of other cells. So, the number of active threads is equal to $4^{splitCount}$, so if we have $splitCount = 5$, then we have a total of 1024 threads. We apply this modification in our Renderer class by creating a new render function called $renderMultithreaded(splitCount)$, and a *renderThread* function which we pass for your thread to run.

In our *renderMultithreaded*, We can define a rectangle (cell) based on the top-left position and bottom-right position of our image. So inside this function, we have:

Listing 10.1: Renderer.h/renderMultithreaded

```
...
uint deltaThread = 1 << splitCount; //2^splitCount
float deltaX = width / float(deltaThread);
float deltaY = height / float(deltaThread);

for (uint y = 0; y < deltaThread; y++) {
    for (uint x = 0; x < deltaThread; x++) {
        uint topX = (uint)deltaX * x;
        uint topY = (uint)deltaY * y;
        uint bottomX = topX + deltaX;
        uint bottomY = topY + deltaY;
    }
}
...
```

Since we now have a dimension of each cell, we will pass these values to our *renderThread* function, which is the same as our original render function, but it runs on a different thread. The signature of this function looks like

Listing 10.2: Renderer.h/renderThread

```
static void renderThread(Renderer* r, uint topX, uint topY,
                         uint bottomX, uint bottomY)
```

We need a reference to our Renderer since we are running on a different then main thread, and we need access to our image buffer. As for the definition, its the same as render, but things like castRay becomes r->castRay.

Listing 10.3: 500x500

```
./renderMain 500 500 0 renderedImage.ppm | Render time: .56 secs
./renderMain 500 500 1 renderedImage.ppm | Render time: .34 secs
./renderMain 500 500 2 renderedImage.ppm | Render time: .31 secs
```

Listing 10.4: 1600x900

```
./renderMain 1600 900 0 renderedImage.ppm | Render time: 2.90 secs
./renderMain 1600 900 1 renderedImage.ppm | Render time: 1.40 secs
./renderMain 1600 900 2 renderedImage.ppm | Render time: 1.04 secs
```

Listing 10.5: 3840x2160 (4k resolution)

```
./renderMain 3840 2160 0 renderedImage.ppm | Render time: 15.72 secs
./renderMain 3840 2160 1 renderedImage.ppm | Render time: 8.15 secs
./renderMain 3840 2160 2 renderedImage.ppm | Render time: 6.60 secs
./renderMain 3840 2160 3 renderedImage.ppm | Render time: 6.46 secs
./renderMain 3840 2160 4 renderedImage.ppm | Render time: 6.07 secs
```

Listing 10.6: 7680x4320 (8k resolution)

```
./renderMain 7680 4320 0 renderedImage.ppm | Render time: 68.30 secs
./renderMain 7680 4320 1 renderedImage.ppm | Render time: 37.22 secs
./renderMain 7680 4320 2 renderedImage.ppm | Render time: 30.26 secs
./renderMain 7680 4320 3 renderedImage.ppm | Render time: 26.41 secs
./renderMain 7680 4320 4 renderedImage.ppm | Render time: 25.66 secs
```

From these runs, we see that having at least 1 split will significantly reduce our run time, and having more does help but not as much.

Chapter 11

Supersampling

After reading this chapter you should be able to:

- Anti-aliasing and one way of doing this
- Jittering and averaging pixels

Whenever we rendered our images, it was sharp and jagged around the edges of our object's edge. Even when we increased our resolution, the problem still showed up. The problem is that we assign each pixel color for 1 ray through that middle of the pixel. This process of smoothing the edges of your objects is called anti-aliasing and is one of the many problems both in ray tracing and rasterization.

To fix this problem, we do something called supersampling. This operation takes place for each pixel, and instead of equaling our pixel value based on our one ray. We instead perform many rays for each pixel and average them for the final pixel color. But averaging the middle point will always give the same point. So instead, we jitter our ray a little and average them instead. This way it gives use nearby information for each pixel. More information on this can be found **Supersampling**. So in our Renderer.h, we define two new variables:

Listing 11.1: Renderer.h

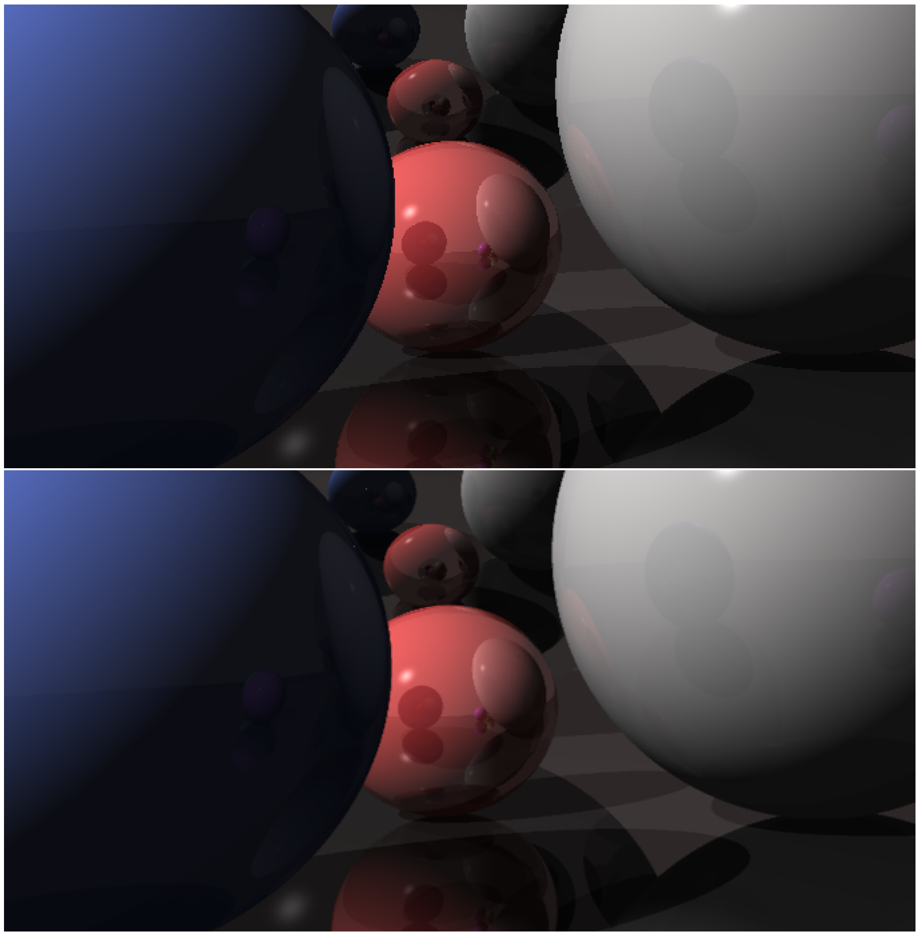
```
uint totalSample = 3;
vector<float> jitterValues{0.25f, 0.75f,-0.75f, 0.33f};
```

And when we call *getRay* in our *render* function. We do an additional loop till totalSample, and accumulate our color values. Also, instead of passing in x and y. We apply shifts from jitterValues. At the end, we divide by totalSample.

Listing 11.2: Renderer.cpp

```
uint bufferIndex = index(x, y);
for (uint sample = 0; sample < totalSample; sample++) {
    float deltaX = jitterValues[2 * sample];
    float deltaY = jitterValues[2 * sample + 1];
    buffer[bufferIndex] += castRay(cameraOrigin,
                           getRay(x + deltaX, y + deltaY), 0);
}
buffer[bufferIndex] /= totalSample;
```

Here is our non-supersampling vs supersampling rendered image:



Chapter 12
Conclusion

Thank you for reaching the end of the book. Whether you read through the book or breezed through it and looked at the images. It means a lot at the end of the day. Hopefully by the end, it, either way, inspired you to learn Ray Tracing (or even Computer Graphics) or to expand on things we didn't go over. But most importantly, you had fun by looking at what is possible and how impressive it is that we can turn pieces of code into these beautiful images. Again, you can get all the code on my GitHub, so you can explore and expand on your own or use it as a reference to build your own. Finally, thank you.