Unity Ray Tracer
Maths and GFX Coursework 4
By Adam Joyce

Introduction

In this project I chose to implement a simple ray tracer in Unity. Below is a high level explanation of my code along with an image of my resulting scene.

C# Files

My project consists of two C# files. The first is 'ObjectRayTracingInfo.cs'. This is where object specific information for the ray tracing is stored, that is values that are used to correctly compute the colour of a pixel dependent on factors such as lighting and reflection. There is only a single 'Awake()' function which simply checks if the object material has a texture and assigns it a default colour is no texture is present.

The second C# file is called 'RayTracer.cs'. This is where the main ray tracing code lives. I will go through the different functions and their purpose below.

Awake()

Similarly to the awake function in 'ObjectRayTracingInfo.cs', this is used for initial administration tasks including fetching all the lights in the scene and setting up the ray traced render texture with the correct width and height according to the resolution and screen.

RayTrace()

This is where the ray tracing process begins. The function cycles through every pixel in scene and creates a ray going from the main camera through those pixel locations. Each ray is used to determine the colour at that pixel, and this colour is then set in the corresponding pixel location in the resultant render texture.

```
27      // Cast rays from the camera to each pixel in the scene and set the render texture pixels accordingly.
28      private void RayTrace() {
29          Color defaultColour = Color.black;
30          for (int x = 0; x < renderTexture.width; x++) {
31              for (int y = 0; y < renderTexture.height; y++) {
32                  Vector3 rayPosition = new Vector3(x / resolution, y / resolution, 0);
33                  Ray ray = GetComponent<Camera>().ScreenPointToRay(rayPosition);
34                  renderTexture.SetPixel(x, y, DetermineColour(ray, defaultColour, 0));
35              }
36          }
37          renderTexture.Apply();
38      }
```

DetermineColour()

'DetermineColour()' is given the ray going from the camera through the current pixel being worked on, a default starting colour, and the raycast current iteration. This iteration value is necessary to avoid infinite computation when two reflective surfaces are reflecting light off one another.

The given pixel's ray is cast to determine if it intersects with any object in the scene. If no collider is hit the default colour of black is returned. If the ray does intersect with an object in the scene the objects' base material or texture colour is added to the cumulative pixel colour.

Once the base colour of the object has been determined the 'HandleLights()' function is called with its result being added to the pixel colours running value. This function takes into account the effects on the pixel's colour from all the different lights in the scene.

After the lighting has been addressed the object's attached 'ObjectRayTracingInfo.cs' script is checked for any reflective or transparent properties. If the object is deemed to be reflective another reflection ray is created and DetermineColour() is recursively called. This resulting reflection colour is multiplied by the object's level of reflection and added to the pixel's cumulative colour value. A similar thing happens if the object shows signs of transparency, with the new ray instead travelling in exactly the same direction as the initial pixel's ray, past the object.

```
40      // Determines the overall colour of the pixel at the location of the ray collision.
41      private Color DetermineColour(Ray ray, Color positionColour, int currentIteration) {
42          if (currentIteration < maximumIterations) {
43              RaycastHit hit;
44
45              // Check the ray intersects a collider.
46              if (Physics.Raycast(ray, out hit, maximumRaycastDistance)) {
47                  // Determine the basic material colour of the pixel.
48                  Material objectMaterial = hit.collider.gameObject.GetComponent<Renderer>().material;
49                  if (objectMaterial.mainTexture) {
50                      Texture2D mainTexture = objectMaterial.mainTexture as Texture2D;
51                      positionColour += mainTexture.GetPixelBilinear(hit.textureCoord.x, hit.textureCoord.y);
52                  } else {
53                      positionColour += objectMaterial.color;
54                  }
55
56                  ObjectRayTracingInfo objectInfo = hit.collider.gameObject.GetComponent<ObjectRayTracingInfo>();
57                  Vector3 hitPosition = hit.point + hit.normal * 0.0001f;
58
59                  positionColour += HandleLights(objectInfo, hitPosition, hit.normal, ray.direction);
60
61                  // Solve reflection pixel colour by casting a new reflection ray.
62                  if (objectInfo.reflectiveCoefficient > 0f) {
63                      float reflect = 2.0f * Vector3.Dot(ray.direction, hit.normal);
64                      Ray newRay = new Ray(hitPosition, ray.direction - reflect * hit.normal);
65                      positionColour += objectInfo.reflectiveCoefficient * DetermineColour(newRay, positionColour, ++currentIteration);
66                  }
67
68                  // Solve transparent pixels by casting a ray from the hit point past the transparent object.
69                  if (objectInfo.transparentCoefficient > 0f) {
70                      Ray newRay = new Ray(hit.point - hit.normal * 0.0001f, ray.direction);
71                      positionColour += objectInfo.transparentCoefficient * DetermineColour(newRay, positionColour, ++currentIteration);
72                  }
73              }
74          }
75          return positionColour;
76      }
```

HandleLights()

The 'HandleLights()' function cycles through each light source in the scene and calculates the light colour on the pixel being processed from all the scene's lighting.

```
78      // Handles the light reflections in the scene.
79      private Color HandleLights(ObjectRayTracingInfo objectInfo, Vector3 rayHitPosition, Vector3 surfaceNormal, Vector3 rayDirection) {
80          Color lightColour = RenderSettings.ambientLight;
81
82          for (int i = 0; i < lights.Length; i++) {
83              if (lights[i].enabled) {
84                  lightColour += LightTrace(objectInfo, lights[i], rayHitPosition, surfaceNormal, rayDirection);
85              }
86          }
87
88          return lightColour;
89      }
```

LightTrace()

'LightTrace()' returns the lighting colour value to 'HandleLight()'. This value is calculated from the light colour, its intensity, and its overall contribution on the pixel being processed.

This function first determines the type of light it is dealing with. Currently my implementation supports Unity's directional and spot lights. Moving forward I plan to expand this to other lights built-in to the engine including the point light.

Once the type of light is known (if the light is not directional) calculations are made to ensure that the pixel being processed falls within range and is actually being lit by the light. Again, depending on

the type of light being addressed, dot product(s) of the light vector and the normal of the surface being lit are taken and used to establish the angle at which the light reflects from the surface. If this angle is greater than zero, and thus the surface is not perpendicular to the light's rays, I cast a ray from the initial hit point location on the object towards the light. This is used to determine if any object is obscuring the initial hit point location thus making it and the pixel on the render texture in shadow. Finally the light contribution is calculated by calling 'CalculateLightContribution()'.

```
91      // Determines the different lighting contributions on a pixel in the scene based on the type of light being processed.
92      private Color LightTrace(ObjectRayTracingInfo objectInfo, Light light, Vector3 rayHitPosition, Vector3 surfaceNormal, Vector3 rayDirection) {
93          Vector3 lightDirection;
94          float lightDistance, lightContribution, dotDirectionNormal;
95
96          if (light.type == LightType.Directional) {
97              lightContribution = 0;
98              lightDirection = -light.transform.forward;
99
100             // Determine the angle that the light reflects of the surface.
101             dotDirectionNormal = Vector3.Dot(lightDirection, surfaceNormal);
102             if (dotDirectionNormal > 0) {
103                 // Returns the colour black if the hit position is in shadow.
104                 if (Physics.Raycast(rayHitPosition, lightDirection, maximumRaycastDistance)) {
105                     return Color.black;
106                 }
107
108                 lightContribution += CalculateLightContribution(objectInfo, dotDirectionNormal, rayDirection, surfaceNormal, light);
109             }
110
111             return light.color * light.intensity * lightContribution;
112         }
113         else if (light.type == LightType.Spot) {
114             lightContribution = 0;
115             lightDirection = (light.transform.position - rayHitPosition).normalized;
116             dotDirectionNormal = Vector3.Dot(lightDirection, surfaceNormal);
117             lightDistance = Vector3.Distance(rayHitPosition, light.transform.position);
118
119             // Ensure the light is within range of the object and the angle of incidence positive.
120             if (lightDistance < light.range && dotDirectionNormal > 0f) {
121                 float dotDirectionLight = Vector3.Dot(lightDirection, -light.transform.forward);
122
123                 // Ensure the object being lit falls within the spot light's radius.
124                 if (dotDirectionLight > (1 - light.spotAngle / 180f)) {
125                     // Returns the colour black if the hit position is in shadow.
126                     if (Physics.Raycast(rayHitPosition, lightDirection, maximumRaycastDistance)) {
127                         return Color.black;
128                     }
129
130                     lightContribution += CalculateLightContribution(objectInfo, dotDirectionNormal, rayDirection, surfaceNormal, light);
131                 }
132             }
133
134             if (lightContribution == 0) {
135                 return Color.black;
136             }
137
138             return light.color * light.intensity * lightContribution;
139         }
140
141         return Color.black;
142     }
```

CalculateLightContribution(), Phong(), BlinnPhong()

'CalculateLightContribution()' uses the object's 'ObjectRayTracingInfo.cs' script values to determine the light's contribution value to a pixel.

Firstly I use Lambert's cosine law to determine the perceived lighting for the current viewing ray when multiplied with the diffuse colour property of the surface.

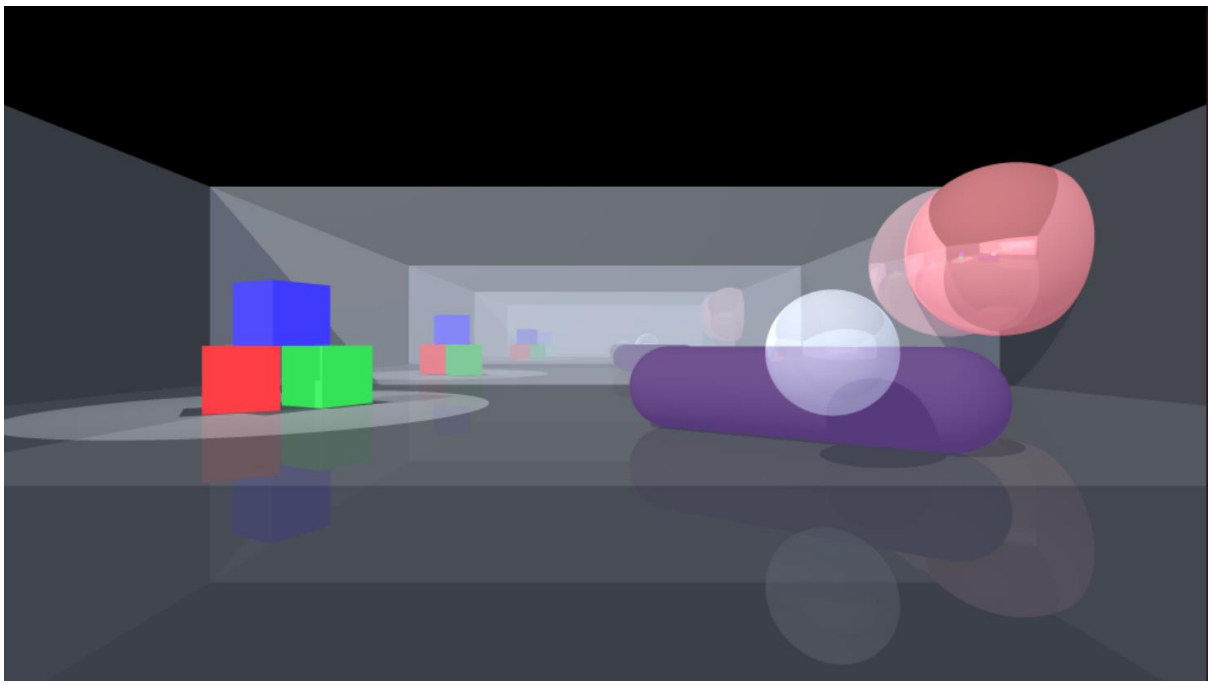
Then, assuming that the object has reflective properties, either the Phong or BlinnPhong process if used. Implementations of these two methods can be found in 'Phong()' and 'BlinnPhong()' respectively.

```
144     // Calculate the light contribuation on a specific object hit point and thus pixel location in the scene.
145     private float CalculateLightContribution(ObjectRayTracingInfo objectInfo, float dotDirectionNormal, Vector3 rayDirection, Vector3 surfaceNormal, Light light) {
146         float lightContribution = 0;
147 
148         if (objectInfo.lambertCoefficient > 0) {
149             lightContribution += objectInfo.lambertCoefficient * dotDirectionNormal;
150         }
151 
152         if (objectInfo.reflectiveCoefficient > 0) {
153             if (objectInfo.phongCoefficient > 0) {
154                 lightContribution += Phong(objectInfo, rayDirection, surfaceNormal);
155             }
156 
157             if (objectInfo.blinnPhongCoefficient > 0) {
158                 lightContribution += BlinnPhong(objectInfo, light, rayDirection, surfaceNormal);
159             }
160         }
161 
162         return lightContribution;
163     }
```

```
165     // Calculate the Phong reflection term.
166     private float Phong(ObjectRayTracingInfo objectInfo, Vector3 rayDirection, Vector3 hitSurfaceNormal) {
167         float reflect = 2.0f * Vector3.Dot(rayDirection, hitSurfaceNormal);
168         Vector3 phongDirection = rayDirection - reflect * hitSurfaceNormal;
169         float phongTerm = Max(Vector3.Dot(phongDirection, rayDirection), 0f);
170         phongTerm = objectInfo.reflectiveCoefficient * Mathf.Pow(phongTerm, objectInfo.phongPower) * objectInfo.phongCoefficient;
171         return phongTerm;
172     }
173 
174     // Calculate the Blinn-Phong reflection term.
175     private float BlinnPhong(ObjectRayTracingInfo objectInfo, Light light, Vector3 rayDirection, Vector3 hitSurfaceNormal) {
176         Vector3 blinnDirection = -light.transform.forward - rayDirection;
177         float temp = Mathf.Sqrt(Vector3.Dot(blinnDirection, blinnDirection));
178         if (temp > 0f) {
179             blinnDirection = (1f / temp) * blinnDirection;
180             float blinnTerm = Max(Vector3.Dot(blinnDirection, hitSurfaceNormal), 0f);
181             blinnTerm = objectInfo.reflectiveCoefficient * Mathf.Pow(blinnTerm, objectInfo.blinnPhongPower) * objectInfo.blinnPhongCoefficient;
182             return blinnTerm;
183         }
184         return 0f;
185     }
```

Result

Below is the resultant image of a simple scene:



Future Development

Beyond the addition of support for more Unity camera types the main objective for me moving forward is to adapt my current code to incorporate real-time rendering functionality. Obviously this would require a number of changes to the code structure and would present an interesting challenge when balancing efficiency with image quality.

Youtube Video

https://youtu.be/kdj3v3epzPM

References

- https://www.ics.uci.edu/~gopi/CS211B/RayTracing%20tutorial.pdf
-
- http://www.cs.cornell.edu/courses/cs4620/2011fa/lectures/08raytracingWeb.pdf
- https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
- https://en.wikipedia.org/wiki/Phong_reflection_model
- http://docs.unity3d.com/ScriptReference/Physics.Raycast.html