Physically Based Renderer

Initial Research

CVA level 5 Computing for Animation

Quentin Corker-Marin
i7624405

Contents

1 Introduction

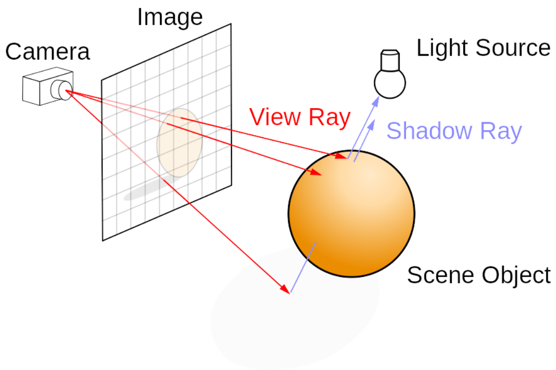

Figure 1: Ray tracing illustration, https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29#/media/File:Ray_trace_diagram.svg

My goal is to implement a physically based rendering system, using a ray-tracing algorithm. The ray tracing algorithm works in the following way:

```
for each pixel in the output image:
        a ray is traced from the camera to that pixel
        the ray is continued into the scene
        if they ray intersects with any geometry
                colour information is calculated at that point by recursively se
                colour information is placed in the pixel that the ray passed th
```

The program flow for the entire render will be the following:

```
init main class
main creates a parser
parser reads scene_desctiption.txt and creates a Scene object
main creates a renderer and passes it the scene description
renderer queues up render tasks
for each render task
        for each pixel in task
                get ray from camera
                send ray into scene
                if ray intersects with geometry
                        calculate shading info from material
                                recursively send out more rays until all require
```

```
                        pass shading info back to task
                else
                        pass background colour back to task
        pass all shading and colour info to film
if remaining tasks == 0
write film information to disk
clean up memory
```

To start with I am planing on implementing the ray tracer to work with triangular meshes, but by having all renderable geometry in the scene inherit from the Shape class and having each implementation of that class handle its own intersection routines, it should be easy to implement other types of primitive geometry such as implicit surfaces.

2 Ray/Triangle Intersection

To calculate intersections between rays and triangles, I will use an algorithm outlined in *Physically Based Rendering, From Theory to Implementation* (Pharr and Humphreys, 2010) that uses barycentric coordinates to parameterise triangles in terms of two variables.

$$p(b_1, b_2) = (1 - b_1 - b_2)p_0 + b_1p_1 + b_2p_2$$

By equating the parametric equation for a ray to the barycentric representation of the triangle, a formula can be derived to get the point of intersection between the ray and the triangle:

$$o + td = (1 - b_1 - b_2)p_0 + b_1p_1 + b_2p_2$$

I don't currently know enough to explain the full derivation. I will need to explore barycentric coordinates to properly understand how to check the intersection.

3 Proposed Classes

I have created a rough outline of the classes that I will need for the renderer implementation.

Main Is responsible for coordinating the program. Once initialised it creates a parser, tells it to parse a .txt file and passes that information to a renderer.

Parser Reads and translates a text file and builds the Scene.

Scene Stores Camera, Lights and Shapes. It can organise the shapes into an acceleration structure for more efficient ray/shape intersection calculations.

Renderer Contains a pointer to the Scene and is responsible for setting up and running a queue of render tasks (for multithreading?).

RenderTask Sends out rays to its designated portion of the final image and is responsible for storing it until the task is finished and the information is sent to the Film.

Film Holds information for the final image and has methods for writing to disk and displaying on screen.

Camera Holds information on Cameras position and attributes (focal length etc.) and when given a set of coordinates of a pixel on the final image, returns the rays required for finding the colour of that pixel.

Light Has a position within the scene and an associated luminance and colour.

Shape The base class for all renderable geometry and acceleration structures to keep the interface the same across all types and allow for easy extensions in terms of primitives that can be rendered and acceleration structures used.

TriangularMesh The first type of renderable geometry that I will implement, it contains an array of triangles.

Triangle Contains 3 points, a pointer to a Material and routines for ray/triangle intersections.

Material Keeps hold of all information required when an intersection is found and shading and colour information is needed.

Vec3 Vector class capable of all necessary vector calculations (dot, cross, add etc.)

Mat3_3 3 by 3 matrix class for transformations. May also need a 4 by 4 matrix for translations.

Ray Holds all necessary information on a ray in the parametric form $o + td$, including its bounce number and current parametric value t.

RGBAColour Container for colour information.

How I believe theses classes will link together is illustrated in a class diagram at the end of this document.

4 BRDF's

Once the intersection point has been calculated, and information about the point of intersection has been found, lighting and colour information needs to be calculated. Part of this requires a Bidirectional Reflectance Distribution Function. For any given viewing direction, the BRDF calculates the relative contribution of each incoming ray.

$$L_o = \int_{\Omega} f(l, v) \bigotimes L_i(l)(n.l) d\omega_i$$

This equation states that the outgoing radiance L_o is equal to the sum over all angles of the incoming radiance $L_i(l)$ times the BRDF $f(l, v)$ times the cosine factor $n.l$

The algorithm for calculating this would be something like:

```
if intersection found:
        total luminance = 0
        for each light in scene
                create new ray between intersection and light
                if ray is not blocked by other geometry
                        luminance += 0
                else
                        luminance += luminance of light * cosine factor * BRDF
```

This is a very brief idea of what I will need to look at, taken from *SIGGRAPH 2010 Course: Physically Based Shading Models in Film and Game Production*

5 Acceleration Structures

After an initial search I have found that a Bounding Volume Hierarchy Would be a good structure to start with. BVH's partition primitive shapes into a hierarchy of disjointed sets. The top node contains the entire scene, and each subsequent node down the tree contains the bounding box for all enclosed primitives. This means that instead of testing for collision with every primitive in the scene, the collision is tested for spaces that contain the primitives, so many primitives can be dismissed with an intersection check of the set that contains them.

The acceleration structure would be a derivative of the Shape class so that it contains the same intersection interface as the other primitive shapes. This make traversing the tree with collision checks easier to implement.

6 Further Reading

The following is a list of material that I have come across and need to explore further.

- Physically Based Rendering, From Theory to Implementation, Pharr and Humphreys
- SIGGRAPH 2010 Course: Physically Based Shading Models in Film and Game Production
- `http://www.cs.utah.edu/~shirley/books/fcg2/rt.pdf`
- NGL facilities

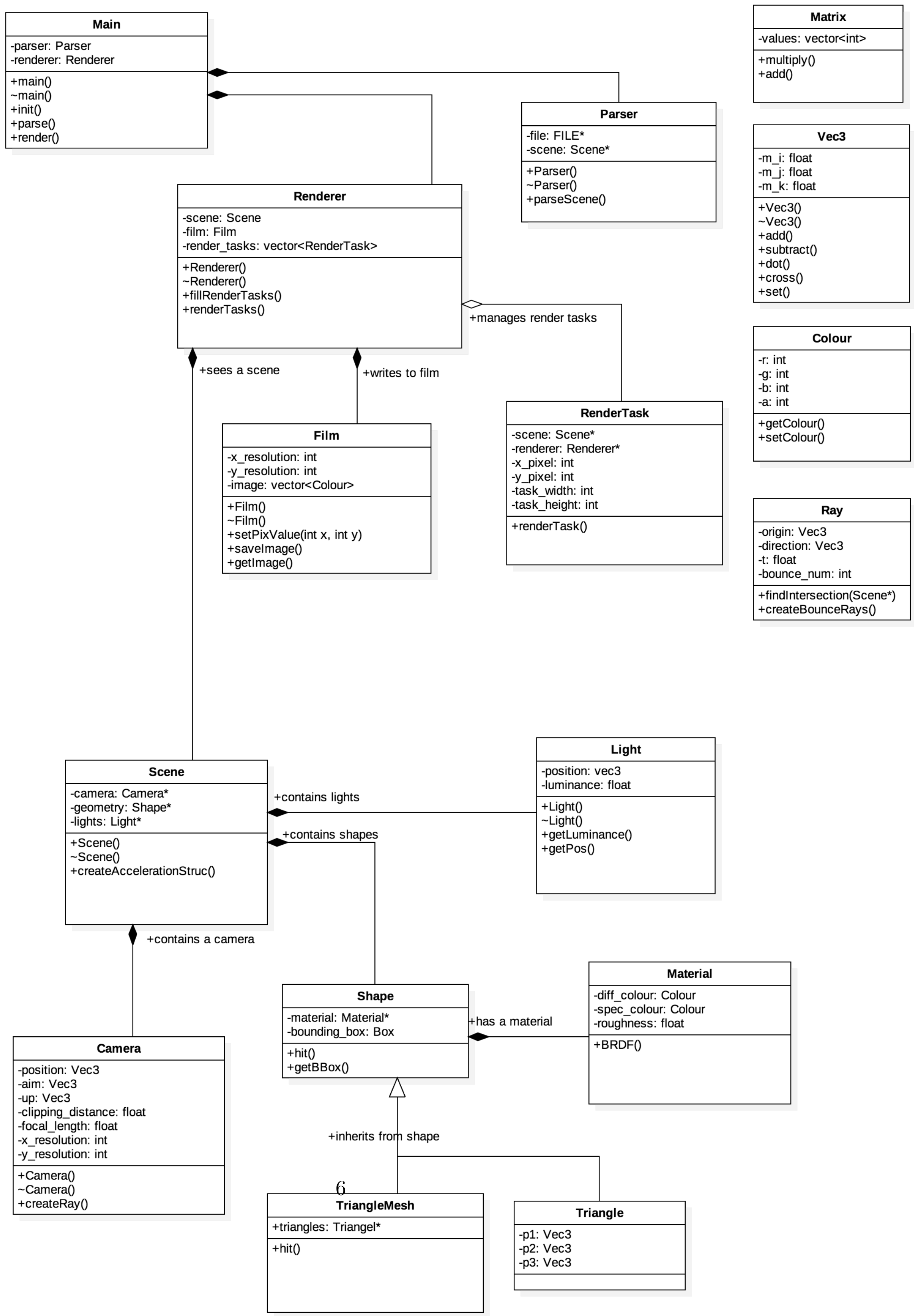

Figure 2: Proposed classes and some of their links for my ray tracer