PHYSICALLY BASED RENDERING

ARDEN RASMUSSEN

NOVEMBER 30, 2019

Abstract. Physically Based Rendering (PBR) is the method for rendering computer generated scenes, in a way that is intended to mimic that of reality. The primary goal of a PBR system, is to simulate every ray of light and its interaction with the scene. The goal of the system that is described in this paper is to make a PBR system with extremely simple user interface, and one that uses the method of ray marching. The library and executable are open source and are available on Github[1].

Part 1. Theory

1. Light Transport I

The primary premise of a physically based render is the simulate the travel of light throughout the scene and its interactions with the objects and materials present. Conceptually consider rays of light leaving a light in a room. As the light bounces around the room it at some point either hits the observers eye, or it bounces away. This basic description is shown in Figure 1.

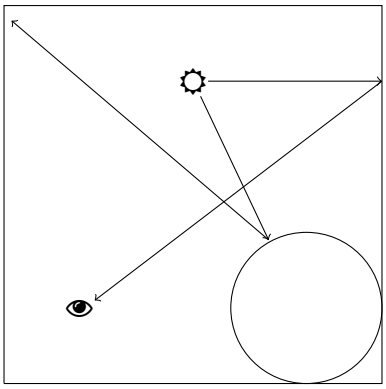


Figure 1. Rays of light propagating from a light source to an observer

From this image it becomes clear that one would need to simulate billions of rays of light for any semblance of an image to form. This is because the observer can be considered a single point in space, then the probability that a ray will travel through an infinitesimal point in the entirety of the region is for all intensive purposes zero. So with this method, we would need to simulate an infinite number of light sources before enough light reaches the observer to construct a representation of the scene.

The way to solve this issue, is to follow the light backwards. The transport of light, in this macroscopic domain, is deterministic, and perfectly reversible. That is to say that if there is an incident ray in direction $\vec{a}$, and it reflects in direction $\vec{b}$, then a light incoming from direction $-\vec{b}$ will be reflected in direction $-\vec{a}$.

The method that rendering systems actually use is to trace the rays of light from the observer into the scene, and then wait until the rays hit a light source. Since the light sources are not point objects, but have a surface, then the probability that a ray will hit a light source could still be small, but it is significantly more probable than a ray hitting the observer. Taking a look at the same scene as before, with this new method is shown in Figure 2.

It is clear that even though not all rays will result in hitting a light source, a significantly larger number will travel between a light source and the observer. Because of this improvement this is the method that we will use for tracing rays of light throughout the scene.

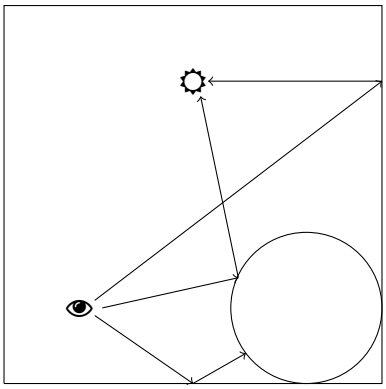


FIGURE 2. Rays of light propagating from an observer to a light source

2. Camera Projection

The first stage of the rendering process is to project the rays from the observer or camera. There are two types of projection, first Orthogonal projection, and the Perspective projection. For the vast majority of situations, the perspective projection is the projection method that will be used, as it more accurately simulates cameras in reality. Because of this we do not go into detail about the orthogonal projection method, nor has it been implemented in Specula.

In more advance physically based renderer, camera lenses and the refractions and effects caused by them are simulated as well. In our system we have not yet implemented the lens system. This means that our camera simulates a pin hole camera, and does not have a field of focus, or any of the features that one would expect from a physical camera.

The model of the camera that we implement is a mirror of the physical pinhole camera. I a pinhole camera, there is the pinhole aperture, and behind that there is the plane of the film. Any of the light that passes through the aperture is projected onto the film. Thus to project light going in the reverse direction, we would consider each point on the film, and project a ray from that point to the point of the pinhole. This process is depicted in Figure 3.

Because of the digital nature of the system, we are able to develop some slight alternatives to the pinhole camera, which are simpler and more efficient to implement. These alternatives are based on moving the plane of the film to be in front of the aperture, in stead of behind it. The distance of the film from the aperture adjusts the field of view of the image. For example if the plane of the film is close to the aperture, then the result would relate to a large field of view, and this case is demonstrated in Figure 3.(b). Then if the field of view is small, this would relate to the film plane being very far away from the plane of the aperture, and this is also demonstrated in Figure 3.(c). This process can be described as selected a point on the film, and projecting the ray of light from the aperture, through this point on the film, and further into the scene.

2.1. **Film.**

For the purposes of generating an image, we will consider the film to be constructed of discrete pixels. By determining the color of each pixel, we can then generate the output image. The data for each pixel will be computed by the

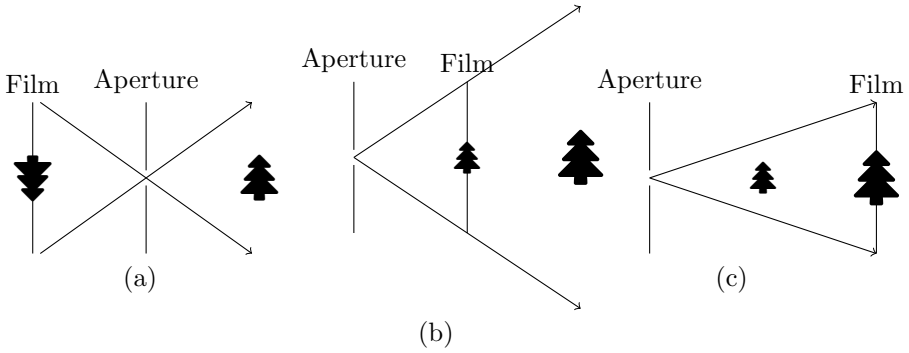


FIGURE 3. A standard pinhole camera(a) produces a flipped image, our camera (b) and (c) produces a upright image, which is proportionally scaled to the scene.

path marching process, and then stored in the film object. Once all pixels have been colors, it is then possible to output the film into any common image format. Specula implements generating `png`, `jpeg`, and `bmp` images.

2.2. **Sampling.** As described previously, the primary process used for determining the colors of the pixels on the film, is to project a ray from the aperture through the film and then into the scene. To do this we must determine which point on the film to project rays of light though.

The naive method to do this, is to send a single ray through the center of each pixel of the film. This will produce acceptable results, but it has some problems. Consider a pixel as shown in Figure 4.(a). This pixel using this method would be grey, the color of the circle. However, that is loosing about 21% of the data in the pixel, all of the white space is being forgotten about, and in this case that accounts for approximately 21% of the pixel, but there are other cases where it could account for more, like in Figure 4.(b), the center pixel will be considered to be grey, but by far the majority of the pixel is white, in this case 87% of the data present in that pixel is being lost.

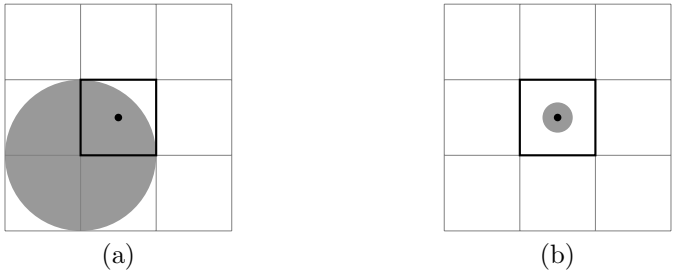
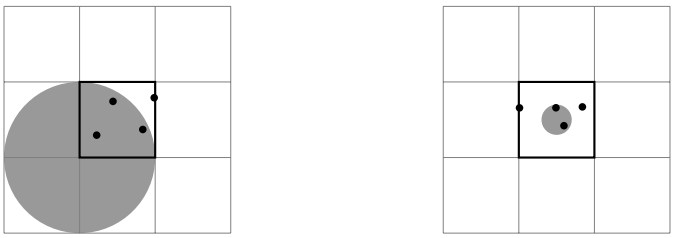


FIGURE 4. Pixel with multiple possibilities

The method that we implement to resolve this issue, is to sample multiple rays for each pixel in the image, where each ray is at a different random location in the pixel. The points in each pixel are uniformly distributed over the area, and thus as

the number of samples per pixel(*spp*) approaches infinity, then the average color of these rays represents the average color of the pixel as a whole.

Considering our examples from before, but by taking multiple samples we preserve more information present in the pixel. In Figure 5.(a), we find that the pixel should be 75% grey, and 25% white. While not the exact value that the color should be, it is a much better representation than a single sample would provide us. In Figure 5.(b), we find that the pixel should be 50% grey, and 50% white. Again, not the exact value that we would expect, but it is significantly better. Both of these examples have only been computed with four samples, and the rendering system will generally sample each pixel in the range of $128 - 4096$, but the user is able to adjust this to be higher or lower as desired.



(a) (b)

FIGURE 5. Pixel with with multiple random samples

Intuitively this is clear that with more samples, the approximation of the color within the pixel becomes more accurate. We verify this using a numeric simulation of the example in Figure 5.(b). It is clear that this process is equivalent to approximating the area of the shaded region within the pixel. So we developed a script to randomly sample points within the pixel to approximate the area of the shaded circle. The resulting plot of the approximated area, and the error of that approximation is shown in Figure 6.

This will be our method for the initial sampling of rays from the camera into the scene, each ray which we sample will return a color. Each ray within a pixel will be averaged and the average color will be used for the color of the pixel.

3. Ray Marching

After the ray has been case from the observer, then next step is to determine what object it hits in the scene, if any. This process is the core the any path tracer or ray tracer.

In most ray tracing or path tracing systems, the preferred implementation is to use a ray tracer. The method that Specula uses is build off of the basis of a ray tracer, so we will explain that first.

For these explanations we will consider a scene with three circles and a plane. This example scene is shown in Figure 7.

3.1. **Ray Tracing.**

Ray tracing proceeds by calculating the rays intersection with all object in the scene. For our example, the ray will intersect with circle A and B. It will actually intersect both of these twice. We will call these point A_1, A_2, B_1, and B_2. Then by sorting these intersection points for which is closest, we find

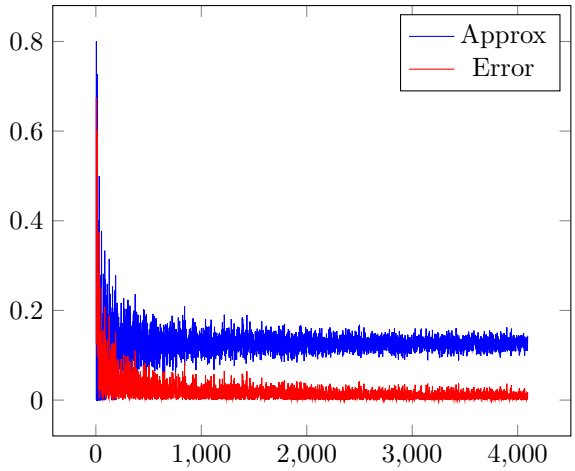


FIGURE 6. Convergence of approximation and error for pixel sampling

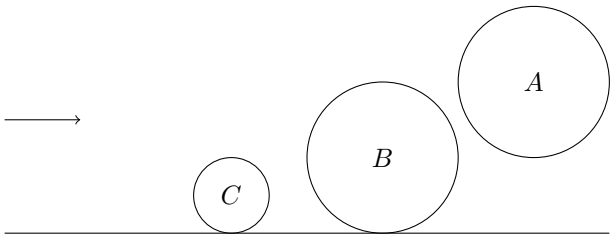


FIGURE 7. Sample scene for explanation

where the intersection point is. This example is depicted in Figure 8, and we notice that the nearest intersection is at point B_1, and we can continue processing from that point.

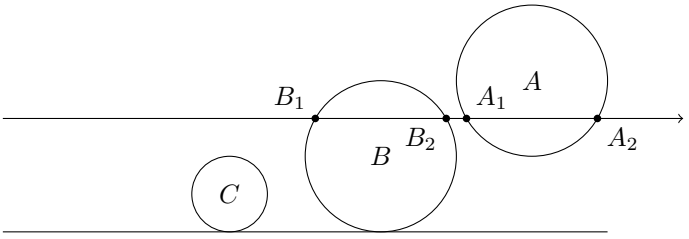


FIGURE 8. Sample scene with ray tracing

To solve for these intersection points we first consider the spherical shell as a collection of points. Thus we can rewrite the equation of a sphere as

$$(x - x_0)^2 + (y + y_0)^2 + (z + z_0)^2 = r^2$$
$$\|P - C\|^2 = r^2.$$

Where P is a point on the sphere, and C is the center point of the sphere, with radius r. We can also consider the equation for the ray as

$$P(t) = A + tB,$$

where A is the origin of the ray, and B is the direction that the ray is traveling in, and t is some scalar denoting the distance along the ray in the direction of B. Combining these two equations we find

$$\|A + tB - C\|^2 = r^2$$
$$\text{dot}\left(A + tB - C, A + tB - C\right) = r^2$$

Expanding this we find

$$\text{dot}\left(B, B\right) t^2 + \text{dot}\left(B, A - C\right) 2t + \text{dot}\left(A - C, A - C\right) - r^2 = 0.$$

Using the quadratic formula we can solve for t, and find

$$t = \frac{-2\text{dot}\left(B, A - C\right) \pm \sqrt{4\text{dot}\left(B, A - C\right)^2 - 4\text{dot}\left(B, B\right)\left(\text{dot}\left(A - C, A - C\right) - r^2\right)}}{2\text{dot}\left(B, B\right)}$$

One we have calculated t, we can determine the point of intersection by computing $A+tB$ with this calculated value for t. A similar process is done for all other shapes in the scene, and then the smallest t value is selected, and that intersection point is used as for further computation.

3.2. **Ray Marching.** The method of ray marching is similar to that of ray tracing, but it never computes the exact intersection point of the ray and an object. The premises of ray marching is to determine how far a ray can travel with the certainty of not intersecting and object. Then the ray moves that far and the process is repeated. When the distance it travels in a step is less than some value, then it is considered as an intersection.

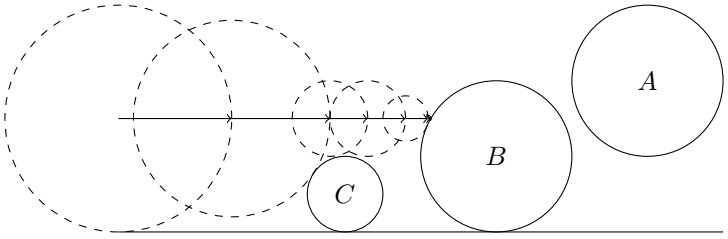


FIGURE 9. Sample scene with ray marching

This process is demonstrated in Figure 9. Any any step, we only need to compute the minimum distance to any object in the scene. This is done recursively until the minimum distance is less than some value ε.

These distance approximations are called distance functions. And ray marching is the method that we will use for our renderer. The reasoning for this is that it provides some features that other methods are unable to simulate. Although it can be slower in the rendering process, if configured correctly it does not cause any degradation to the resulting image.

4. Distance Functions

For the ray marching method, each object must have a function that given a point returns the distance from that point to the object. In this section we will provide distance functions for many primitives, and some formulas for combining primitives together into more complex shapes. Even more complex shapes can have a distance function such as more complex fractals.

4.1. **Primitives.** All primitives are centered at the origin, any scaling translation or rotation must be done separately.

4.1.1. *Sphere.*

```
return length(p)-radius;
```

4.1.2. *Box.*

```
vec3 q = abs(p) - b;
return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
```

4.1.3. *Round Box.*

```
vec3 q = abs(p) - b;
return length(max(q,0.0)) + min(max(q.x,max(p.y,q.z)),0.0) - r;
```

4.1.4. *Torus.*

```
vec2 q = vec2(length(p.xz)-t.x,p.y);
return length(q) - t.y;
```

4.1.5. *Plane.*

```
return dot(p,n.xyz)) + n.w;
```

4.1.6. *Cylinder.*

```
vec2 d = abs(vec2(length(p.xz),p.y)) - vec2(h,r);
return min(max(d.x, d.y), 0.0) + length(max(d, 0.0));
```

4.1.7. *Rounded Cylinder.*

```
vec2 d = vec2(length(p.xz)-2.0*ra+rb, abs(p.y)-h);
return min(max(d.x, d.y), 0.0) + length(max(d, 0.0)) - rb;
```

4.1.8. *Cone.*

```
vec2 q = vec2( length(p.xz), p.y );
vec2 k1 = vec2(r2,h);
vec2 k2 = vec2(r2-r1,2.0*h);
vec2 ca = vec2(q.x-min(q.x,(q.y<0.0)?r1:r2), abs(q.y)-h);
vec2 cb = q - k1 + k2*clamp( dot(k1-q,k2)/dot2(k2), 0.0, 1.0 );
float s = (cb.x<0.0 && ca.y<0.0) ? -1.0 : 1.0;
return s*sqrt( min(dot2(ca),dot2(cb)) );
```

4.1.9. *Round Cone.*

```
vec2 q = vec2( length(p.xz), p.y );

float b = (r1-r2)/h;
float a = sqrt(1.0-b*b);
float k = dot(q,vec2(-b,a));

if( k < 0.0 ) return length(q) - r1;
if( k > a*h ) return length(q-vec2(0.0,h)) - r2;

return dot(q, vec2(a,b) ) - r1;
```

4.1.10. *Triangle.*

```
vec3 ba = b - a; vec3 pa = p - a;
vec3 cb = c - b; vec3 pb = p - b;
vec3 ac = a - c; vec3 pc = p - c;
vec3 nor = cross( ba, ac );

return sqrt(
(sign(dot(cross(ba,nor),pa)) +
sign(dot(cross(cb,nor),pb)) +
sign(dot(cross(ac,nor),pc))<2.0)
?
min( min(
dot2(ba*clamp(dot(ba,pa)/dot2(ba),0.0,1.0)-pa),
dot2(cb*clamp(dot(cb,pb)/dot2(cb),0.0,1.0)-pb) ),
dot2(ac*clamp(dot(ac,pc)/dot2(ac),0.0,1.0)-pc) )
:
dot(nor,pa)*dot(nor,pa)/dot2(nor) );
```

4.1.11. *Quad.*

```
vec3 ba = b - a; vec3 pa = p - a;
vec3 cb = c - b; vec3 pb = p - b;
vec3 dc = d - c; vec3 pc = p - c;
vec3 ad = a - d; vec3 pd = p - d;
vec3 nor = cross( ba, ad );

return sqrt(
(sign(dot(cross(ba,nor),pa)) +
sign(dot(cross(cb,nor),pb)) +
sign(dot(cross(dc,nor),pc)) +
sign(dot(cross(ad,nor),pd))<3.0)
```

```
  ?
  min( min( min(
  dot2(ba*clamp(dot(ba,pa)/dot2(ba),0.0,1.0)-pa),
  dot2(cb*clamp(dot(cb,pb)/dot2(cb),0.0,1.0)-pb) ),
  dot2(dc*clamp(dot(dc,pc)/dot2(dc),0.0,1.0)-pc) ),
  dot2(ad*clamp(dot(ad,pd)/dot2(ad),0.0,1.0)-pd) )
  :
  dot(nor,pa)*dot(nor,pa)/dot2(nor) );
```

4.2. **Alterations.** It is possible to apply operation that will change the shape of the primitives, while retaining the exact euclidean metric to them.

4.2.1. *Rounding.*

```
return primative(p) - radius;
```

4.2.2. *Onion.*

```
return abs(primative(p)) - thickness;
```

4.3. **Combinations.** These function combine, carve, or intersect the different primitives. These function can take the result of any two distance functions we will denote these values as `d1` and `d2`.

4.3.1. *Union.*

```
return min(d1,d2);
```

4.3.2. *Subtraction.*

```
return max(-d1,d2);
```

4.3.3. *Intersection.*

```
return max(d1,d2);
```

4.3.4. *Smooth Union.*

```
float h = clamp( 0.5 + 0.5*(d2-d1)/k, 0.0, 1.0 );
return mix( d2, d1, h ) - k*h*(1.0-h);
```

4.3.5. *Smooth Subtraction.*

```
float h = clamp( 0.5 - 0.5*(d2+d1)/k, 0.0, 1.0 );
return mix( d2, -d1, h ) + k*h*(1.0-h);
```

4.3.6. *Smooth Intersection.*

```
float h = clamp( 0.5 - 0.5*(d2-d1)/k, 0.0, 1.0 );
return mix( d2, d1, h ) + k*h*(1.0-h);
```

4.4. **Mandelbulb.** Using the format of a distance function it is possible to generate more complex objects, like this example is the distance function to render the mandelbulb.

```
vec3 z = p;
float dr = 1.0;
float r = 0.0;
for (int i = 0; i < Iterations ; i++) {
  r = length(z);
  if (r>Bailout) break;

  // convert to polar coordinates
  float theta = acos(z.z/r);
  float phi = atan(z.y,z.x);
  dr =  pow( r, Power-1.0)*Power*dr + 1.0;

  // scale and rotate the point
  float zr = pow( r,Power);
  theta = theta*Power;
  phi = phi*Power;

  // convert back to cartesian coordinates
  z = zr*vec3(sin(theta)*cos(phi), sin(phi)*sin(theta), cos(theta));
  z+=p;
}
return 0.5*log(r)*r/dr;
```

Part 2. Practice

5. Outline

In practice the renderer is organized into several steps, these steps are briefly outlined below, and the structure diagram is in Figure.10.

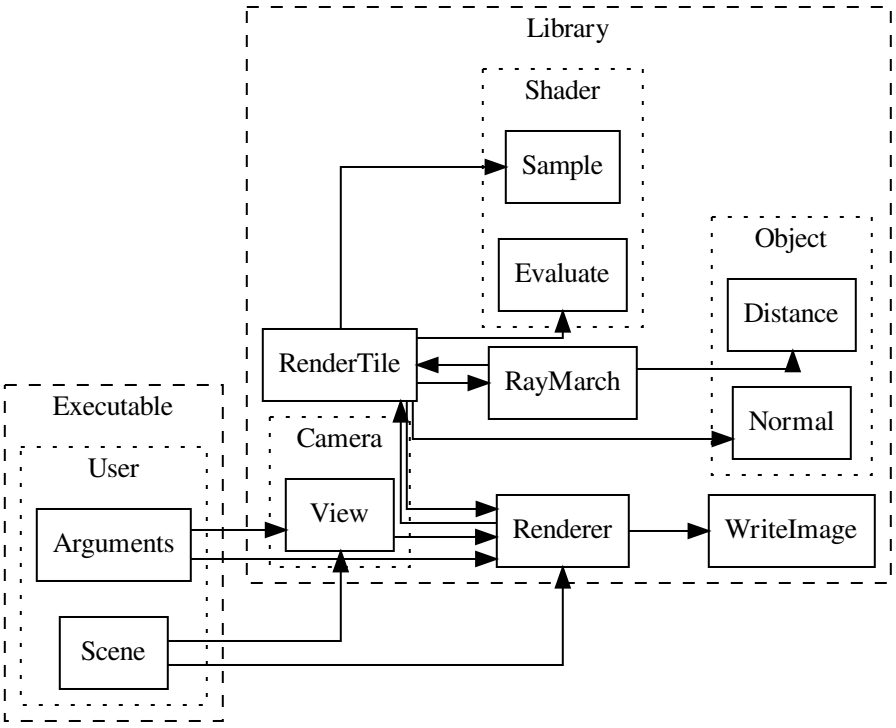


Figure 10. Renderer Process Outline

(1) *User* These are the parts of the process that are defined and controlled by the user. These stages in the process only interact with the camera's view, and the renderer.
   (a) *Arguments* These are the command line arguments specified by the user at runtime.

   (b) *Scene* This is the scene definition script defined by the user.
(2) *Camera* These processes control and alter the camera in the scene.
   (a) *View* This controls the cameras view, including its position, orientation, and projection.
(3) *Renderer* This is the core process of the renderer. It handles constructing the scene, spawning the subprocesses and constructing the output files.
(4) *RenderTile* This process is the core of the renderer, it handles detecting the intersections, processes the shading, and spawning recursive rays to march.
(5) *RayMarch* This process detects the nearest intersection point of a ray in the scene, it is integral for the ray marching to function.
(6) *Object* These stages are defined for every object, and define what the object is, these are the core of the ray marching process.
   (a) *Distance* This is the core part of *Ray Marching*, this function defines the distance from any point to the given object.
   (b) *Normal* This is simply used to numerically approximate the normal of the surface of an object at a given point.
(7) *Shader* These stages define the look of each object, and are at the core of the path tracing process.
   (a) *Sample* This stage is used to greatly improve the efficiency of path tracing, is selects the reflected/refracted ray based upon importance sampling from the material definition.
   (b) *Evaluate* This process evaluates the color of light traveling along the view ray, due to the sampled incident ray.

6. User Control

The user controls were a critical component of the Specula library. One of the key goals of the library was to be easily used, and remove as many complications as possible from the process of creating a scene. The balance from providing the user with all the power provided by a path marcher, while at the same time creating a simple system was remedied, by allowing the user direct access to the core library, and developing an interface system that handles most of the boilerplate code.

6.1. **Command Line Arguments.** This is the first method that the user will interact with the renderer, is through the command line arguments. To develop this, Specula makes use of an open source library CLI11[2]. This library provides a generic command line argument parser, and several validators that the code makes use of. The validators are used to restrict the types of values that can be entered, thus restricting the arguments to sensible values.

The CLI11 library provides most of the validators, such as an `CLI::ExistingFile` and `CLI::PositiveNumber` validators. However, for some arguments a more generalizable validator was required, so using the inheritance of the base validator (`CLI::Validator`), the implementation of a `RegexValidator` was made. The regex validator takes a regular expression at construction, and will test the users input against that pattern, and if the input fails the match the pattern, then an error is thrown.

Parsing the command line arguments is the first step that happens in the entirety of the code, this way if an argument is invalid then the program exists before

[2] `https://github.com/CLIUtils/CLI11`

processing anything. Once these arguments have been parsed, then they can be used throughout the rest of the process, by any component in the executable. It is important to note that is is entirely separated from all of the library code. This means that it is possible to utilize the library without including this system for parsing command line arguments.

The specifics of the command line arguments are further described in Section 7.

6.2. **Scene Description.** The primary form of interface that the user will have with the library, is with the scene definition script. The scene definition script is specified at run time, and is a `Lua` script that the user uses to define the scene's geometry and properties.

Having the scene defined in a `Lua` script, means that all of `Lua`'s standard libraries are available to the user, and the scene description can be generated at runtime through the script. The choice to use `Lua` was primarily based upon its ease of integration with `C/C++`, and its performance. `Lua` usually executes faster than most scripting languages, and there are further improvements that can be made to improve the performance even more.

To integrate the existing libraries API with the Lua scripting language, we use two open source tools. Firstly LuaJIT[3] is a *Just In Time* compiler for lua. This means that at execution time, the lua code is compiled into byte code for improved efficiency and performance. Secondly we use Sol2[4]. Sol2 is a library that abstracts the interface with the lua interpreter provided by LuaJIT. With these too tools, it is possible to provide almost all of the available function from the `C++` API to the lua interface. This means that all of the object primitives that have been defined for use, are also available as classes in the Lua interpreter.

The key difference between the `C++` interface, and the `Lua` interface, is that when the Lua version of the `render()` is called, then the process recognizes that the intention is to render a sequence of images, and handles that accordingly. If `render()` is never called in the scene description script, then the process assumes that a single image was intended to be rendered, and renders the state of the scene after completing of the scene description execution. In the `C++` interface it is required that the `render()` function is explicitly called.

Similarly to the command line arguments, all of the scene description and the `Lua` scripting is contained within the executable process, and will not directly interact with the library. This means that only the executable requires the LuaJIT and Sol2 libraries, which are of significant size, thus reducing the size of the core Specula library.

The specifics of the API that is available in the scene description script is explained in Section 8.

[3] `http://luajit.org/`

[4] `https://github.com/ThePhD/sol2`

Part 3. Analysis

Part 4. Usage

7. Command Line Arguments

The command line arguments can be used to define global scene parameters. These options primarily specify the output of the program, and can also control some global parameters for the renderer.

7.1. **Options.** These options do not effect the renderer in any way, and are used for debugging information, and to get a help message.

`-h,--help`**:** Displays built in help message.

`-v`**:** Changes the verbosity of the output. Include more of this flag for a more verbose output. The range is from $0 - 6$, where 6 will print all levels of messages.

7.2. **Output.** These options directly effect what image files are generated, and the type and size of these output images.

`--aspect RATIO`**:** Defines the aspect ratio of the image to generate. This must be of the form `w:h`, or `wxh`, any other formats will cause an error. Here `w` and `h` can be either floating point number or integer numbers, but must be positive.

`-a,--albedo`**:** Enabling this flag will generate additional files with the file path based upon the output file path `"file-albedo.extension"`, which contain the unshaded colors of the scene. This is primarily useful for denoising the image after generation.

`-d,--depth`**:** Enabling this flag will generate additional files with the file path based upon the output file path `"file-depth.extension"`, which contain the depth map of the scene. This is primarily useful for denoising the image after generation.

`-n,--normal`**:** Enabling this flag will generate additional files with the file path based upon the output file path `"file-normal.extension"`, which contain the normal directions of the scene. This is primarily useful for denoising the image after generation.

`-o,--output FILE`**:** Output file and directory. If a single frame is being rendered, then this will be the path to the file. If multiple frames are being rendered, then this argument will be the directory to generate the numbered files in and the extension for each file. Valid extensions to generate are `png`, `jpeg`, and `bmp`, these are the only extensions that have been defined for the image generator, all other extensions will cause an error.

`-r,--res,--resolution WIDTH`**:** This defines the output resolution width of the image. Combined with the usage of `--aspect`, this allows the construction of any size image file.

7.3. **Renderer.** These options effect the renderer, and can greatly improve or hurt the performance or can change how the output looks.

`-b,--bounces NUM`: This variable controls the minimum number of bounces that the renderer will simulate. More bounces will produce a more realistic result, but will also significantly slow down the simulation. Note that this only defines the minimum number of bounces, most rays will bounce $5 - 10$ times more than this value. This defaults to 10.

`-f,--fov FOV`: This defines the horizontal field of view for the generated image. The vertical field of view is calculated based upon the aspect ratio. For this argument `FOV` must be a floating point number, representing the radians for the field of view. This argument defaults to 45°.

`--no-denoise`: This flag will disable the built in denoiser. Note that has not yet been implemented, no denoiser will be used.

`-s,--spp SAMPLES`: This will determine the number of samples to render for each pixel of the image. A higher value produces significantly better resulting images, but will greatly increase the render time. This value defaults to 16.

`-t,--tile SIZE`: This argument defines the size of the tile, and consequently the maximum number threads to use. Each rendered image is divided into multiple tiles, and this argument defines the size of the tile. This defaults to 16, meaning that each tile is 16×16 pixels.

8. Scene File

All of the scene files are scripted using Lua. All of Lua's syntax is applicable to the scene definitions. In addition to any standard Lua libraries, Specula also provides additional types and functions. These functions and types are what should be used to construct the scene specification.