PHYSICALLY BASED RENDERING

ARDEN RASMUSSEN

NOVEMBER 29, 2019

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

Part 1. Theory

Part 2. Practice

1. Outline

In practice the renderer is organized into several steps, these steps are briefly outlined below, and the structure diagram is in Figure.1.

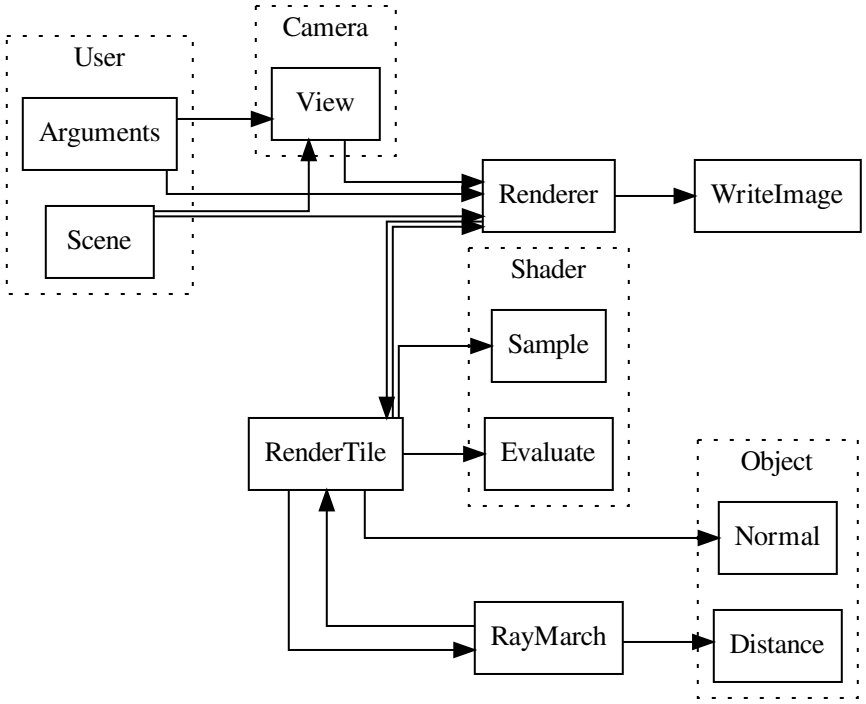

Figure 1. Renderer Process Outline

(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.

Part 3. Analysis

Part 4. Usage

2. 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.

2.1. 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.

2.2. 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.

2.3. **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.

3. 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.