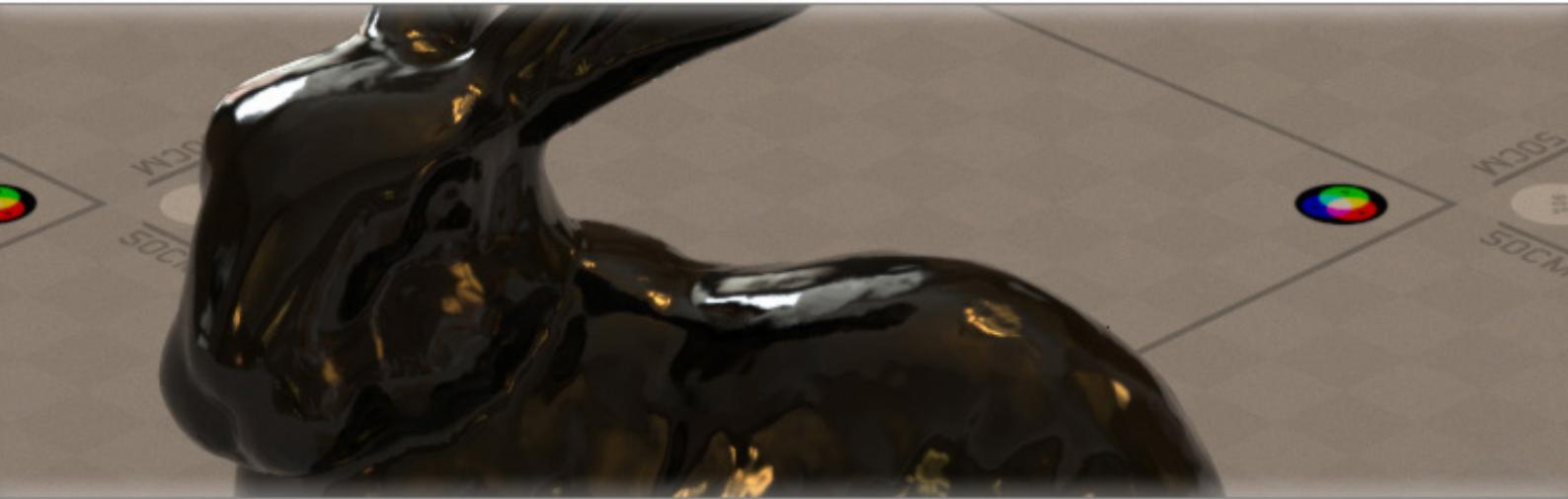


Appendix A

User Guide



KOI

renderer

User Manual

Contents

I. Overview	4
1. Introduction	5
1.1. About Koi Renderer	5
1.2. Intended use	5
1.3. Overview of Features	6
1.3.1. General	6
1.3.2. Materials	6
1.3.3. Lights	6
1.3.4. Cameras	6
1.3.5. Integrators	6
1.4. Limitations	7
II. Installing	8
2. System Requirements	9
2.0.1. Windows	9
3. Compiling	10
3.1. Compiling and Building the Renderer Library	10
3.2. Compiling and Building the Viewer	11
4. Using Koi Renderer	12
4.1. User Interface	12
4.1.1. Preview Window	13
4.1.2. Toolbar	13
III. Getting Started	14
5. Overview	15
6. File Format	16
6.1. Commonly used elements	16
6.1.1. Colour	17
6.1.2. Transform	17
6.1.3. Albedo	18
7. General Settings	19
7.1. Common	19
7.2. Sampler	20
7.2.1. Random	20
8. Cameras	21
8.1. Perspective	21

9. Geometry	22
9.1. Alpha Map	23
9.2. Obj	24
9.3. Quadrilateral	25
10. Materials	26
10.1. Diffuse	27
10.2. Blinn	28
10.3. Plastic	29
10.4. Rough Refractive	30
10.5. Reflective	31
10.6. Refractive	32
10.7. Ward	33
10.8. Blend	34
10.9. Emissive	35
10.10 Two-sided	36
11. Lights	37
11.1. Point	38
11.2. Spot	39
11.3. IES	40
11.4. Area	41
11.5. Dome	42
12. Volume	43
12.1. Phase	44
12.1.1. Isotropic	44
12.1.2. Henyey-Greenstein	44
12.2. Homogeneous	45
12.3. Grid	47
13. Integrator	50
13.1. Whitted	51
13.2. Path	52
13.3. Photon	53

Part I.

Overview

1. Introduction

1.1. About Koi Renderer

Koi is a rendering engine that uses physically based techniques, such as area light sources, linear workflow, and BRDFs, to render realistic images. The key idea behind the renderer is to provide a simple framework that can be understood completely in a short amount of time.

C++ has been used to developed the rendering engine, and currently compiles into a statically-linked library, which exposes all the functionality. Another important aspect of the renderer is, that it does not depend on any external libraries making it easy to compile and use.

The Viewer program uses the Koi's function and relies on Qt a cross platform GUI library, which is used to display the progress on a render, so that the user does not have to wait until it finishes to be able to view it, which may take many minutes.

The development of this engine has been inspired by many other renderers including:

- V-Ray (<https://www.vray.com/>)
- PBRT (<http://pbrt.org/>)
- Takua Render (<http://www.yiningkarlli.com/>)
- Mitsuba Renderer(<https://www.mitsuba-renderer.org/>)
- SmallPT (<http://www.kevinbeason.com/smallpt/>)

1.2. Intended use

The rendering engine is intended for educational purposes rather than in a production environment, therefore readability of code has been given priority over performance. The engine provides a core, but minimal, set of features that allows one to either render images or easily extend it.

1.3. Overview of Features

1.3.1. General

- Physically correct
- High dynamic range imaging (HDRI)
- Instancing
- Render previewer

1.3.2. Materials

- Fresnel reflections
- Lambertian BRDF
- Ward's anisotropic BRDF
- Reflective/Refractive BRDF
- Microfacet BRDFs
- Normal mapping
- Alpha mapping
- Blend material
- Two-sided material
- Emissive material

1.3.3. Lights

- Point light
- Spot light
- Area lights
- IES profile light
- Dome light
- Distant disc light
- Distant directional light

1.3.4. Cameras

- Orthographic camera
- Perspective camera
- Custom bokeh shape
- Depth of field

1.3.5. Integrators

- Normal
- Occlusion
- Whitted ray tracing
- Pathtracer
- Photon mapper
- Volumetric rendering
 - Voxel grid
 - Constant

1.4. Limitations

There are currently quite a few limitations with the rendering engine that the user should be aware of:

- Although Koi supports a variety of textures, only the bitmap and constant colour textures are made available to the XML description language.
- Normal maps work with all materials, but they have only been added to the plastic material.
- The renderer does not support spectral rendering.
- There is no multiple importance sampling so renders are slow to converge.
- The photon mapper only supports diffuse and refractive materials.
- Only PPM and PFM image formats are supported.
- Only performs random sampling.
- The application has only been tested on the Windows platform.

Part II.

Installing

2. System Requirements

2.0.1. Windows

Before using the renderer make sure the system meets the minimum requirements. Currently Koi only supports 64 bit operating systems.

OS	Windows 7/8 (64 bit)
Processor	Intel or AMD CPU
RAM	4GB
Storage	30MB available space

3. Compiling

You can choose run the Windows executable, or choose to compile either the renderer library or viewer yourself if you need to target another platform or Qt version.

Visual Studio 2013 and above is required to compile the renderer. To compile the viewer Qt version 5.5 and above must be installed, and the Windows SDK is required for developing Qt on the Windows platform. The Qt Add-in for visual studio 2013 must be installed to develop Qt using Visual Studio.

You may need to download the Nov 2013 CTP visual studio compiler if you are using the 2013 version of Visual Studio, because some of the source code uses C++11.

Lastly, once the project has been loaded make sure that the solution configuration is set to release. The debug build is much slower.

Visual studio can be downloaded from the link below:

<https://www.visualstudio.com/>

There is a comprehensive step-by-step guide for installing Qt for a windows machine in the link below:

<http://doc.qt.io/qt-5/windows-support.html>

The link to the Qt5 Add-in for Visual Studio and can be found below

<http://www.qt.io/download-open-source/#section-7>

Once everything has been installed and setup. You can now begin the process of compiling. First make sure the solution you want to compile is the active one by right clicking on the solution and clicking on "Set up as StartUp project".

3.1. Compiling and Building the Renderer Library

1. On the menu bar left click "Build".
2. When the submenu is visible left click "Build Solution".
3. Once the solution has been compiled and built this should produce a *statically linked library* file named "PhotonMapping.lib" in the root project folder under x64\release\ folder.
4. You need this link the library file and include all the associated header files to be able to start using them in your own project.
 - a) Add the directory with the all the header files under
Configuration Properties->VC++ Directories
in the field named Include Directories.
 - b) Add the directory with the *.lib file to
Configuration Properties->VC++ Directions
in the field named Library Directories.
 - c) Link the library (*.lib) file by going to
Configuration Properties->Linker->Input->Additional Dependencies ,and
add the name of the (*.lib) of the library e.g. PhotonMapping.lib.

3.2. Compiling and Building the Viewer

1. On the menu bar left click "Build".
2. When the submenu is visible left click "Build Solution".
3. This will produce an executable `Viewer.exe` with the associated configuration and platform.
4. The viewer relies on three Qt dynamic runtime libraries (*.dll); `Qt5Core.dll`, `Qt5Gui.dll`, and `Qt5Widgets.dll`. You can compile to another version if you so wish.
5. These files are listed under your Qt installation directory. You must use the correct version depending on which version of visual studio you are using and what version of Qt you are compiling under. There is a folder `msvc2013_64` for Visual Studio 2013 64 bit
 - a) `Qt\%version%\msv2013_64\bin\Qt5Core.dll`
 - b) `Qt\%version%\msv2013_64\bin\Qt5Gui.dll`
 - c) `Qt\%version%\msv2013_64\bin\Qt5Widgets.dll`
 - d) Copy these files to the directory the `Viewer.exe` is in.

4. Using Koi Renderer

This section provides an overview of the user interface, the following part will describe the file format used by the previewer to render a scene, and details on all available statements.

4.1. User Interface

The user interface consist of a toolbar and preview window labelled 1 and 2, respectively in screenshot below.

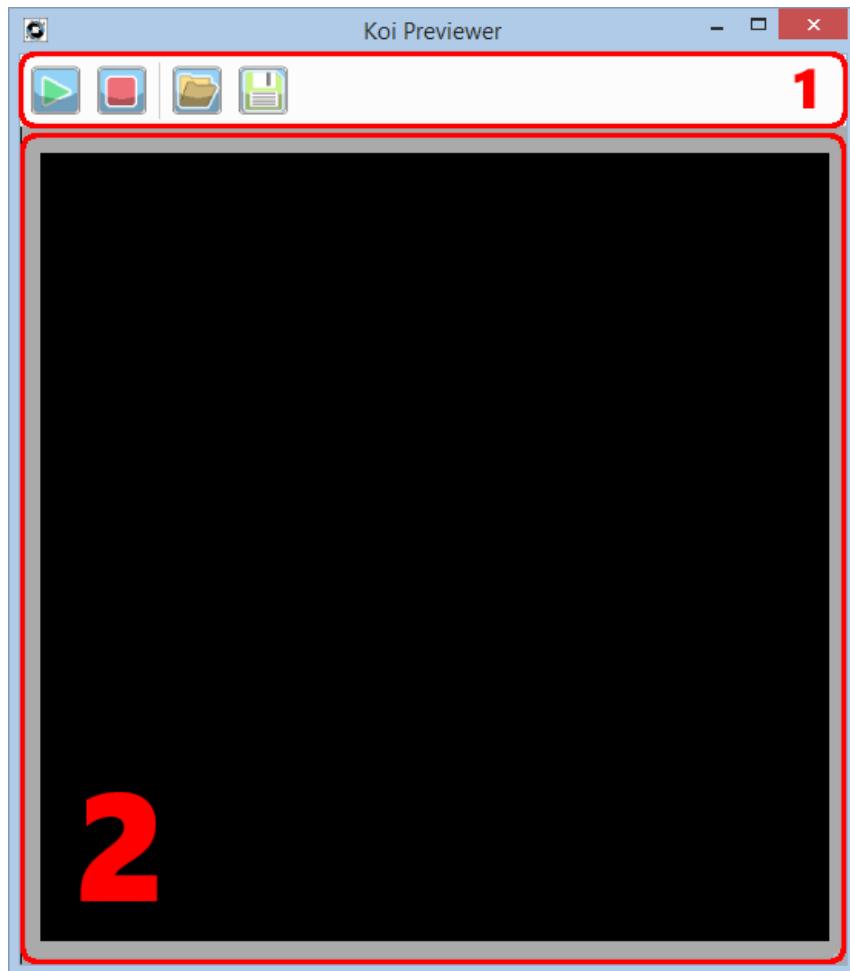


Figure 4.1.: The GUI to preview renders.

4.1.1. Preview Window

The preview window displays the current progress of the render to avoid waiting until it finishes rendering to view it. The window also contains a few features to allow the previewed image to be zoomed in and panned. Mouse wheel: Zoom in/out of window Alt + Left mouse button: Pan the window R: Reset to default position

4.1.2. Toolbar

The main functionality of the previewer is accessed from the toolbar.

Rendering a Scene



Begins rendering the currently loaded scene and displays the current progress on the render window.

Opening a Scene



Opens a new scene from a XML scene description file. Once loaded the scene replaces currently open scene, if any.

Stopping a Render



Stops the current render in progress. The partial render can be saved to disk.

Saving a Render



The save button allows the current progress of a render to be stored on disk as a common image format.

Part III.

Getting Started

5. Overview

The following chapters will provide a detailed look into the XML schema used to define a scene. It will describe each attribute for the available elements and will provide examples of usage. The layout for each chapter is split into a short description of the base class where applicable. Subsequent sections will describe each of the available *types* and their associated attributes, and list isolated XML code to define the syntax.

6. File Format

Koi uses XML, a markup language, to provide a description of the scene to be rendered. The description of the scene must be enclosed within the root element containing a scene element.

Listing 1: root scene element

```
1 <scene>
2     <!-- scene description -->
3     ...
4 </scene>
```

XML elements that correspond with classes in C++ with inheritance hierarchies or elements with many choices will have a *type* attribute specifying the particular instance.

Listing 2: General scene element with inheritance

```
1 <camera type="...">
2     <!-- specific camera elements -->
3     ...
4 </camera>
```

The majority of attributes pertaining to elements use two different structures. One structure involves an element with only one attribute in which case an element-value pair is used. The second case an element may take on many attributes and so uses the name of the field followed by its corresponding value. As an example consider the case defining a volume, which uses both structures. The elements *albedo*, *sigmaA*, *sigmaS*, and *emission* defined using the first type, whereas *phase*, *bbox*, and *maya* are defined using the second.

Listing 3: Defining the attributes of elements

```
1 <volume type="homogeneous" stepSize="0.1">
2     <albedo value="1 0 0"/>
3     <sigmaA value="0.20 0.20 0.20"/>
4     <sigmaS value="0.00 0.00 0.00"/>
5     <emission value="0 0 0"/>
6     <phase type="hg" g="0.0" />
7     <bbox min="-25 0.01 -25" max="25 50 25" />
8     <transform>
9         <maya translation="0 43.372 0" rotation="0 0 0"/>
10    </transform>
11 </volume>
```

6.1. Commonly used elements

There are some commonly used elements that are used over and over again therefore to avoid confusion they will be defined in the following sections.

6.1.1. Colour

The colour element represents a RGB colour value.

Attribute	Type	Default value	Description
value	Colour	white	Specifies the red, green, and blue components in that order.

Listing 4: Defining common settings using XML

```
1 <!-- defines the colour red -->
2 <colour value="1 0 0"/>
```

6.1.2. Transform

The transform element represents a 3D-affine transformation. It is used to place and size objects. There are two types of transformations *max*, and *maya*, which correspond to the differences in coordinate spaces used by default in these software packages.

Attribute	Type	Default value	Description
translation	Vector	(0, 0, 0)	Specifies how much to move in the x, y, and z axes respectively.
rotation	Vector	(0, 0, 0)	Specifies the rotation in the x, y, and z axes in that order.

Listing 5: Defining common settings using XML

```
1 <!-- defines the colour red -->
2 <colour value="1 0 0"/>
```

6.1.3. Albedo

The diffuse albedo gives the overall reflectance of a material. In short it defines the colour of an object and is represented using a value for red, green, and blue. You can either specify the rgb values or use an image file.

Attribute	Type	Default value	Description
value	Colour	-	Specifies the values of the red, green, and blue channels in that order.
bitmap	Bitmap	None	Specifies the location of the image file.

Listing 6: Defining common settings using XML

```
1  <albedo value="r g b"/>
2  <!-- or -->
3  <albedo bitmap="filepath"/>
```

7. General Settings

7.1. Common

The global settings that affects the entire scene such as the final output resolution is defined by the ***common*** element.

Attribute	Type	Default value	Description
width	int	800	Defines the number of horizontal pixels in the output image
height	int	600	Defines the number of vertical pixels in the output image

Listing 7: Defining common settings using XML

```
1 <common>
2   <!-- Defines the width and height of the output image -->
3   <width value="512"/>
4   <height value="512"/>
5 </common>
```

7.2. Sampler

Samplers define the global number of rays to cast per pixel and also the type of sampler to use, currently only random sampling is supported. A larger number of samples reduces the noise in the image, but also increases the rendering times.

7.2.1. Random

The random sampler uses pseudo random numbers to generate samples.

Attribute	Type	Default value	Description
sampleCount	int	16	Defines the number of samples per pixels

Listing 8: Defining a random sampler using XML

```
1 <sampler type="random">
2   <sampleCount value="16"/>
3 </sampler>
```

8. Cameras

The camera determines what part of the scene is in view by specifying the camera's position and orientation in the world. There are three types of cameras, orthographic, perspective, and thin, but only the perspective camera has been made available to the XML description file.

8.1. Perspective

The perspective camera simulates a pinhole camera and has an infinite depth of field, the position and orientation is defined using the transform element, and its zoom is affected by the field of view.

Attribute	Type	Default value	Description
transform	Transform	Identity	Defines the camera to world transformation.
fov	float	35.0	Defines the vertical field of view.

Listing 9: Defining a perspective camera using XML

```
1 <camera type="perspective">
2   <transform>
3     <!-- use either lookat, max, or maya attribute -->
4     <lookat target="0 0 -1" origin="0 0 0" up="0 1 0"/>
5     <max translation="0 0 0" rotation="0 0 0"/>
6     <maya translation="0 0 0" rotation="0 0 0"/>
7   </transform>
8   <fov value="15.805"/>
9 </camera>
```

9. Geometry

Koi allows both analytic geometry and triangle meshes. They are used to represent the outer surface of object. Koi currently supports, *quadrilateral*, *triangle mesh*, *sphere*, and *disc* geometric shapes, but only the first two have been made available to the viewer.

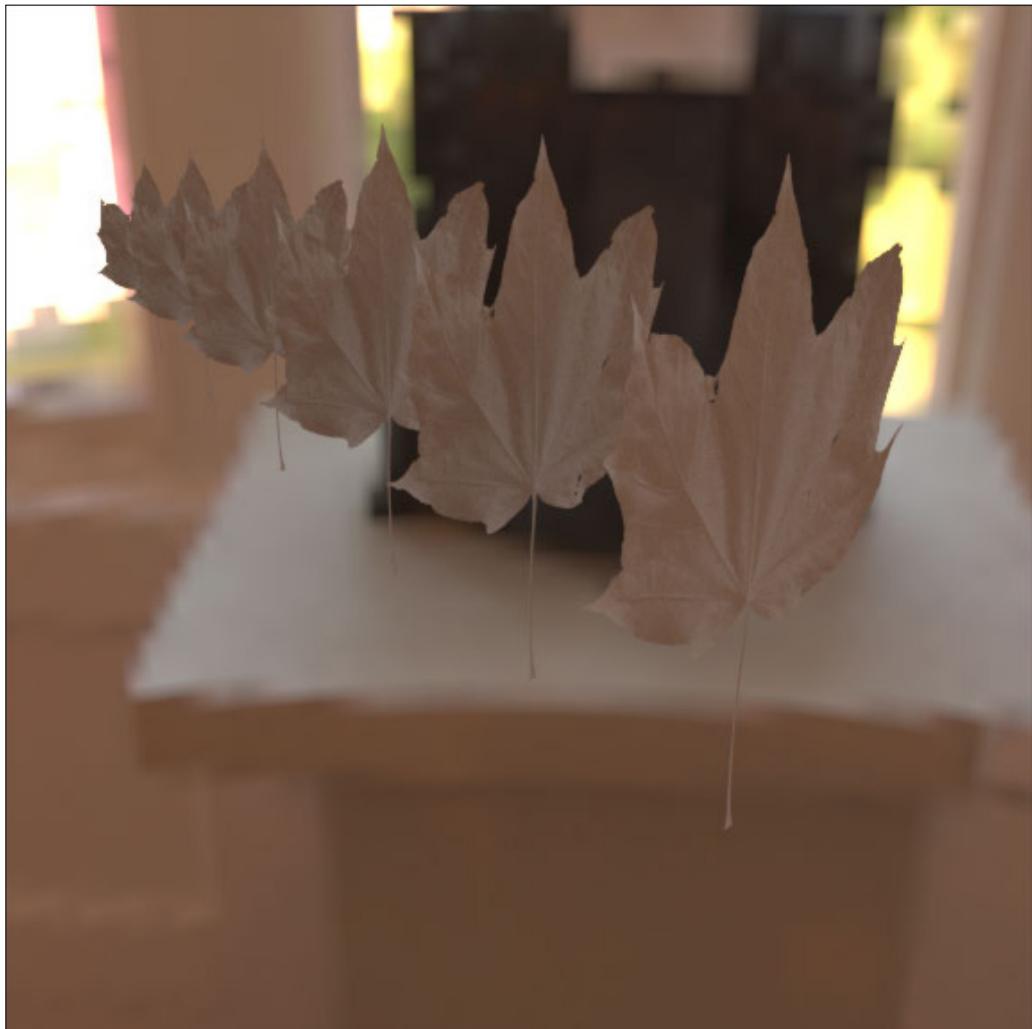
9.1. Alpha Map

Alpha mapping uses a texture to mask parts of the object making certain parts of it transparent. This technique is useful for leaves and cutting shapes through thin geometry, which would otherwise have resulted in more complex geometry.

Attribute	Type	Default value	Description
bitmap	Bitmap	None	The location of the image to be used as a mask.

Listing 10: Defining a alpha map within geometry element

```
1 <geometry type="geometry type">
2   ...
3   <alpha bitmap="filepath"/>
4   ...
5 </geometry>
```



(a) Rendering of leaves using an alpha map

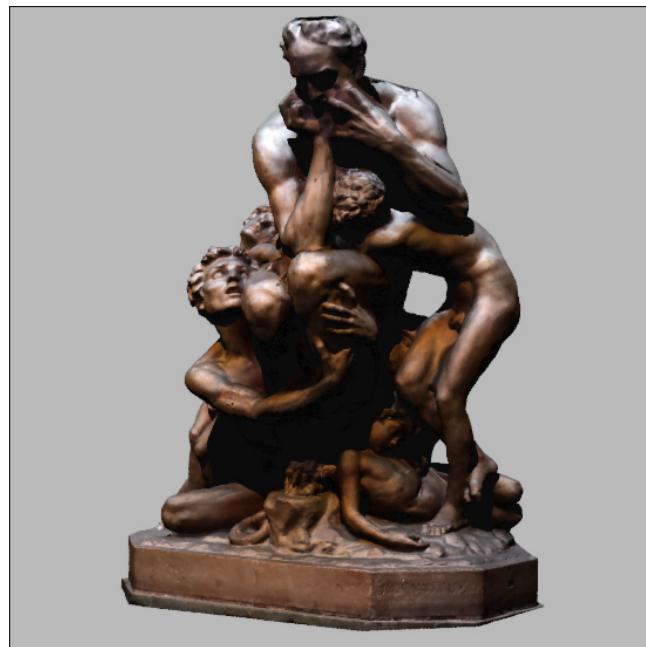
9.2. Obj

The OBJ is a file format used to represent surfaces, and can be used to represent very complex objects. Koi currently only supports triangulated data at the moment.

Attribute	Type	Default value	Description
file	Geometry	None	The location of *.OBJ file to represent the surface.
alphaMap	Bitmap	None	The location of the image to be used as an alpha mask.
transform(opt)	Transform	Identity	This gives the local to world transformation to place the object in the world.
Material	Material	None	The light scattering properties defining the appearance of the object.

Listing 11: Defining a obj in XML

```
1 <geometry type="obj">
2   <file value="path to \*.obj"/>
3   <material type="type">
4     <!-- material definition -->
5   </material>
6   <transform>
7     <!-- transform definition -->
8   </transform>
9 </geometry>
```



(a) Rendering of complex geometry, the model created by Geoffrey Marchal

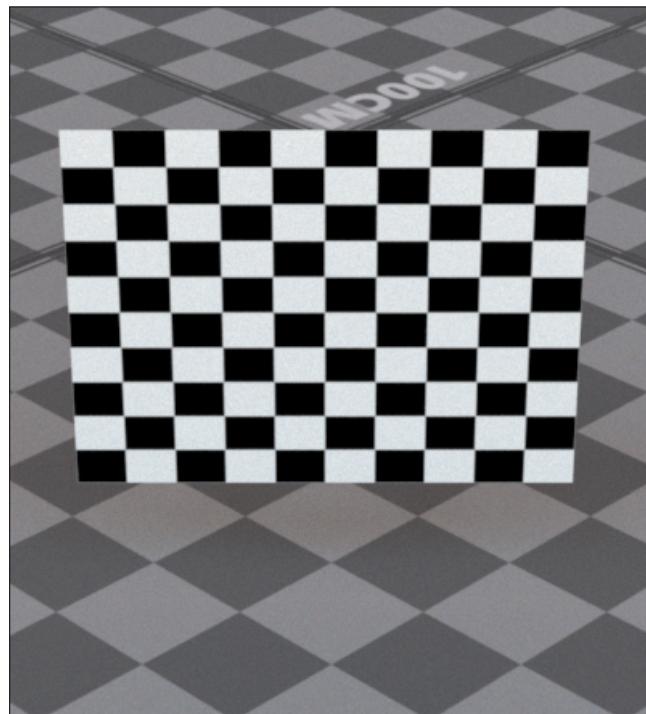
9.3. Quadrilateral

The element `quadrilateral` provides an analytic shape representing a rectangular polygon with four edges. These are useful as the geometric input to area lights.

Attribute	Type	Default value	Description
file	Geometry	None	The location of *.OBJ file to represent the surface.
alphaMap	Bitmap	None	The location of the image to be used as an alpha mask.
transform(opt)	Transform	Identity	This gives the local to world transformation to place the object in the world.
Material	Material	None	The light scattering properties defining the appearance of the object.

Listing 12: Defining a quadrilateral in XML

```
1 <geometry type="quadrilateral">
2   <width value="65" />
3   <height value="52.5" />
4   <transform translation="278 -279.6 547.8" rotation="0 0 0" />
5   <material type="type">
6     <!-- material definition -->
7   </material>
8 </geometry>
```



(a) Rendering of a quadrilateral geometry

10. Materials

The materials in Koi represent the physically properties of a surface, they describe how light is reflected when a ray interacts with it. This section will go through all the materials available, list each of the attributes, and show the effect it has on the visual appearance as they varied.



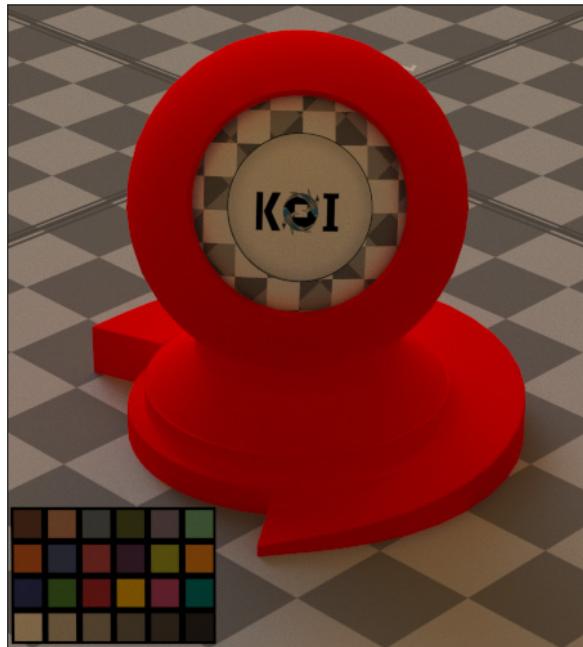
10.1. Diffuse

A *diffuse* material reflects light uniformly in all directions giving it a matte appearance. A few real world materials such as chalk, matte paint, etc. approximates this scattering model. The underlying model used is the Lambertian BRDF.

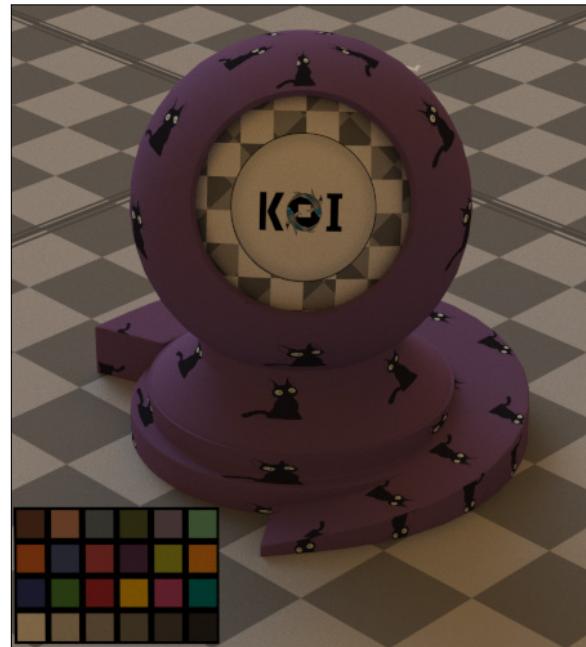
Attribute	Type	Default value	Description
albedo	Colour/Bitmap	white	Specifies the colour/reflectance of the material.

Listing 13: Defining a diffuse material using XML

```
1 <material type="diffuse">
2   <!-- uses a solid colour -->
3   <albedo value="1 1 1"/>
4   <!-- loads an image -->
5   <albedo bitmap="filepath"/>
6 </material>
```



(a) Rendering of material ball with a red diffuse colour



(b) Rendering of material ball with a texture map

10.2. Blinn

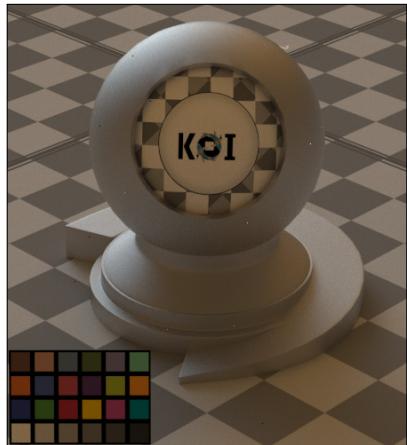
The *blinn* material reflects the light with a bias towards the reflection direction with an exponential falloff. The blinn material is good for modelling unpolished metallic surfaces with blurred highlights.

Attribute	Type	Defaults	Description
albedo	Colour/Bitmap	white	Specifies the colour/reflectance of the material.
ior	float	1.5	Specifies the index of refraction.
roughness	float	1000	Specifies how rough/reflective the material is. A higher value corresponds to a more reflective appearance.

Listing 14: Defining a Blinn material using XML

```

1 <material type="blinn">
2   <albedo value="1 1 1"/>
3   <albedo bitmap="filepath"/>
4   <ior value="10"/>
5   <roughness value="250"/>
6 </material>
```



(a) Blinn roughness=10



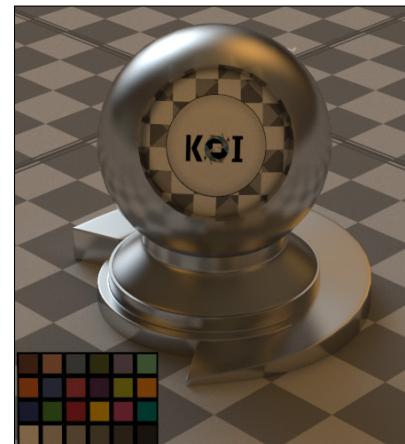
(b) Blinn roughness=20



(c) Blinn roughness=50



(d) Blinn roughness=100



(e) Blinn roughness=500



(f) Blinn roughness=1000

10.3. Plastic

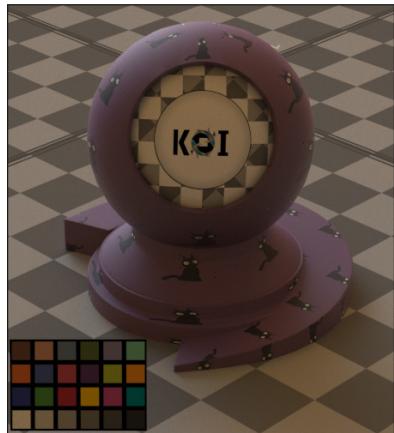
The *plastic* material has a diffuse component and a specular coating. This material should be used to model general materials such as laminate flooring, stone, ceramics, and plastics etc.

Attribute	Type	Defaults	Description
albedo	Colour/Bitmap	white	Specifies the colour/reflectance of the material.
specular	Colour	white	Defines the Colour of specular highlight.
normal	Bitmap	None	The Location of image specifying a normal map.
ior	float	1.5	Specifies the index of refraction.
roughness	float	1000	Specifies how rough/reflective the material is. A higher value corresponds to a more reflective appearance.

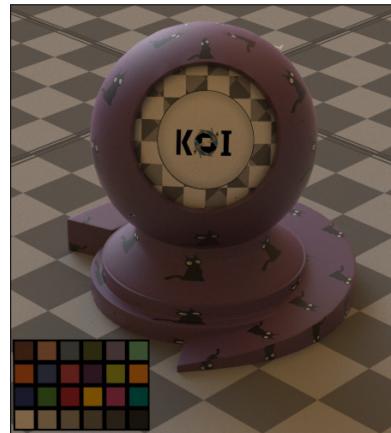
Listing 15: Defining a plastic material using XML

```

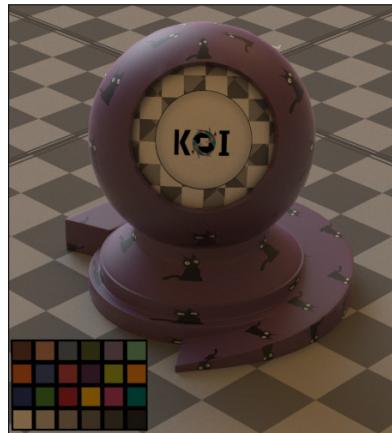
1 <material type="plastic">
2   <albedo bitmap="filepath" tileU="5" tileV="5"/>
3   <normal bitmap="filepath"/>
4   <ior value="1.6"/>
5   <roughness value="1000"/>
6   <specular value="1 0 0"/>
7 </material>
```



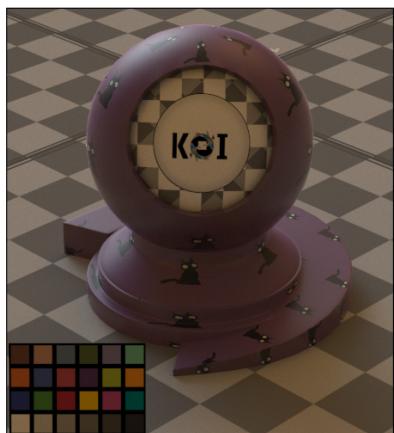
(a) Plastic roughness=10



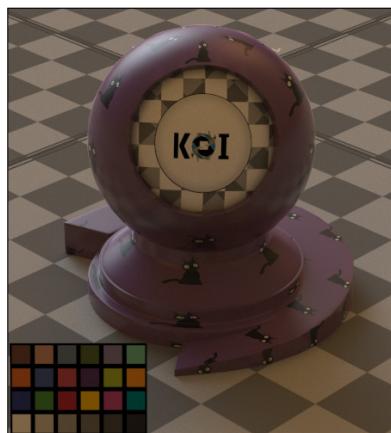
(b) Plastic roughness=20



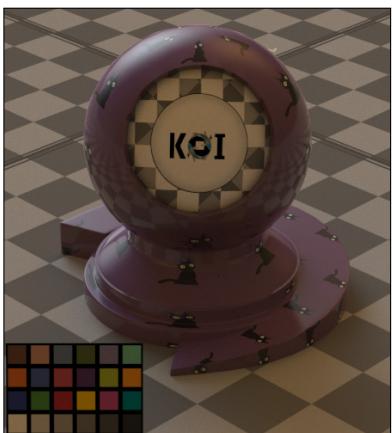
(c) Plastic roughness=50



(d) Plastic roughness=100



(e) Plastic roughness=500



(f) Plastic roughness=1000

10.4. Rough Refractive

The *rough refractive* material gives the appearance of a glossy and rough transparent surface. This is good to model objects such as frosted glass/plastic.

Attribute	Type	Defaults	Description
albedo	Colour/Bitmap	white	Specifies the colour/reflectance of the material.
specular	float	1.5	Defines the Colour of specular highlight.
ior	float	1.5	Specifies the index of refraction.
roughness	float[0-1]	1	Specifies how rough/reflective the material is. A higher value the rougher/glossy the appearance.

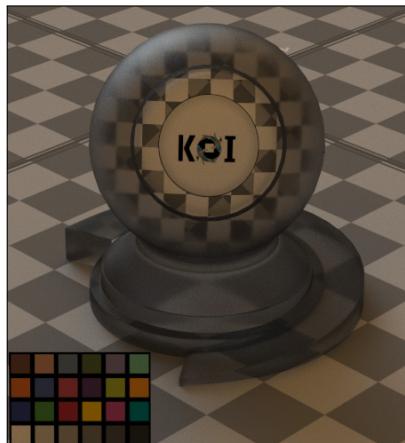
Listing 16: Defining rough refractive material using XML

```

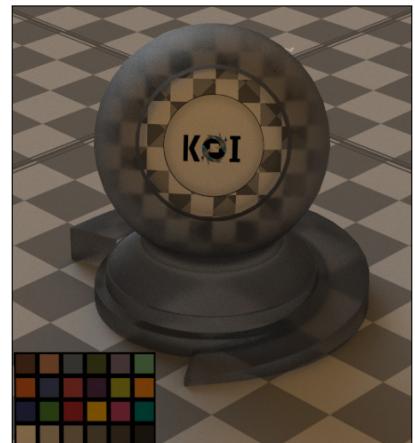
1 <material type="roughRefractive">
2   <ior value="1.6" />
3   <roughness value="0.8" />
4 </material>
```



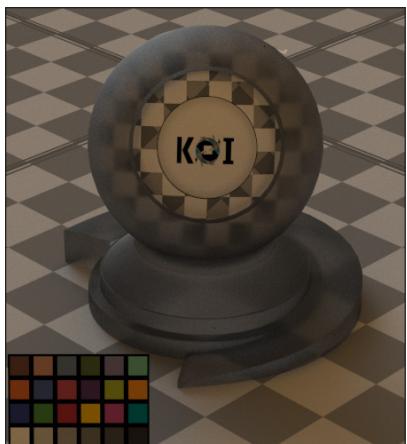
(a) Parameter roughness=0.0



(b) Parameter roughness=0.1



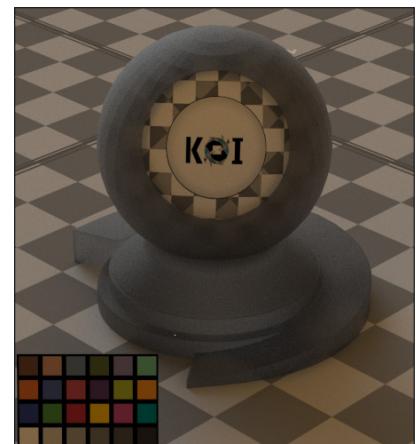
(c) Parameter roughness=0.2



(d) Parameter roughness=0.3



(e) Parameter roughness=0.5



(f) Parameter roughness=1

10.5. Reflective

The *reflective* material models a perfect mirror that reflects light with an angle equal to the incoming light. It is useful to model highly polished metals or glass mirrors.

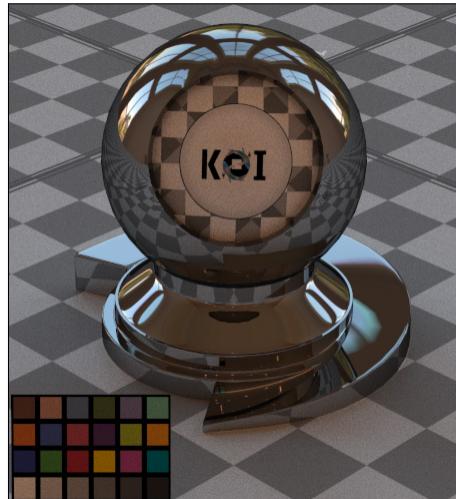
Attribute	Type	Defaults	Description
colour	Colour	white	Specifies the colour of the material.
ior	float	1.5	Specifies the index of refraction.
k	float	50.0	Specifies the absorption coefficient.

Listing 17: Defining a refractive material using XML

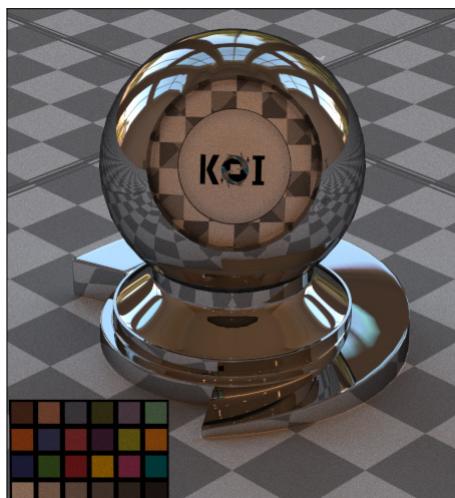
```
1 <material type="reflective">
2   <colour value="1 1 1"/>
3   <k value="25.0"/>
4 </material>
```



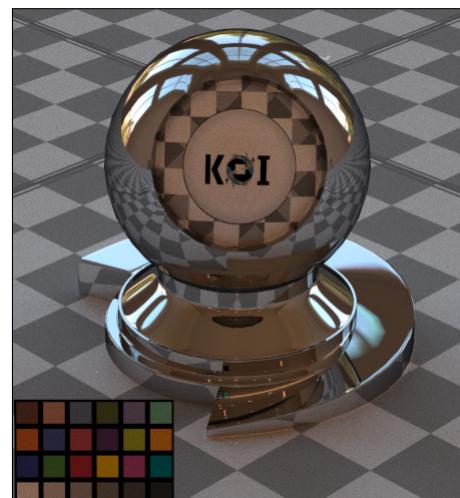
(a) $\text{ior}=1.5, \text{k}=0$



(b) $\text{colour}=\text{white}, \text{ior}=1.5, \text{k}=2.5$



(c) $\text{ior}=1.5, \text{k}=5.0$



(d) $\text{ior}=1.5, \text{k}=7.5$

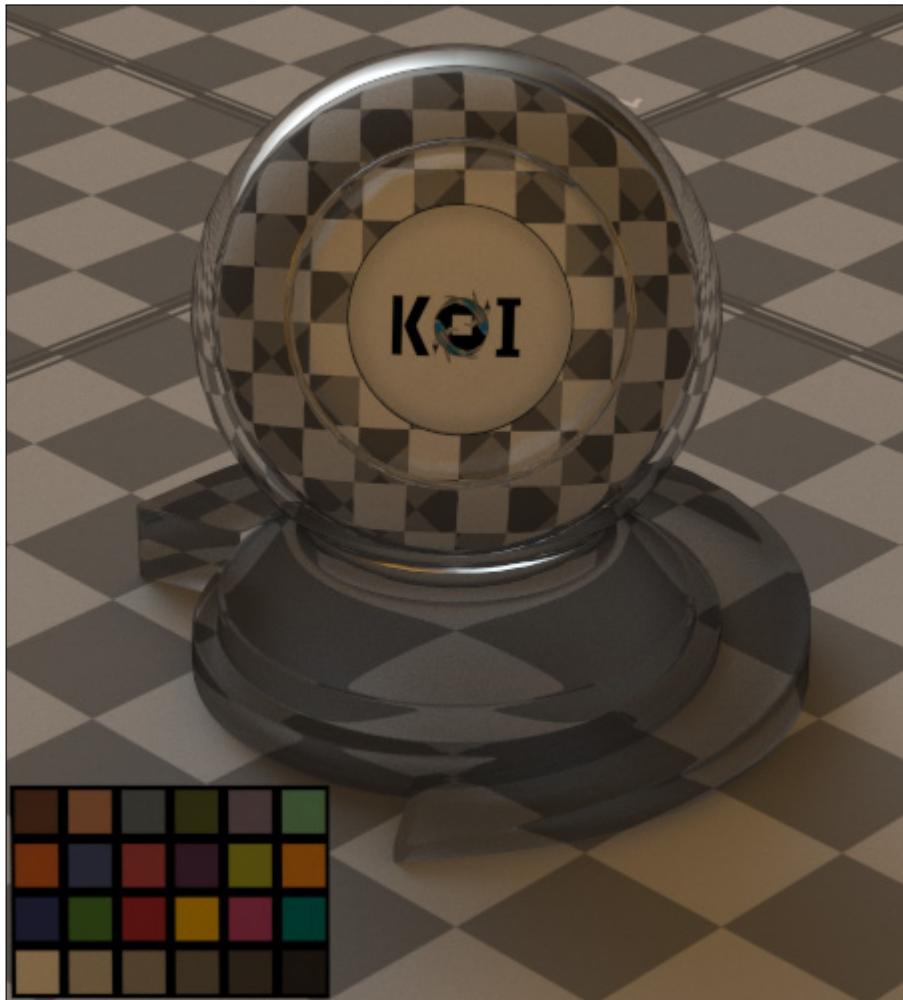
10.6. Refractive

The *refractive* material models transparent objects that transmit and bend light. It is useful for clear glass.

Attribute	Type	Defaults	Description
colour	Colour	white	Specifies the colour of the material.
ior	float	1.5	Specifies the index of refraction.

Listing 18: Defining a reflective material using XML

```
1 <material type="refractive">
2   <colour value="1 1 1"/>
3   <ior value="1.5"/>
4 </material>
```



(a) Rendering of a refractive material ior=1.5

10.7. Ward

The *Ward* material is anisotropic giving stretching of the highlights and rotations about the normal. This material can model surfaces such as brushed metal.

Attribute	Type	Defaults	Description
ax	float[0-1]	0.5	Specifies the anisotropy along U direction.
ay	float[0-1]	0.5	Specifies the anisotropy along V direction.
ior	float	1.5	Specifies the index of refraction.
specular	Colour	white	Defines how reflective the material is.
anisotropy	Bitmap	None	The location of an image defining how anisotropy the surface is.

Listing 19: Defining a Ward material using XML

```
1 <material type="ward">
2   <ax value="0.5"/>
3   <ay value="0.5"/>
4   <ior value="1.6" />
5   <specular value="1 1 1"/>
6   <anisotropy bitmap="filepath" tileU="4" tileV="4" />
7 </material>
```



(a) Ward with image driving anisotropy

10.8. Blend

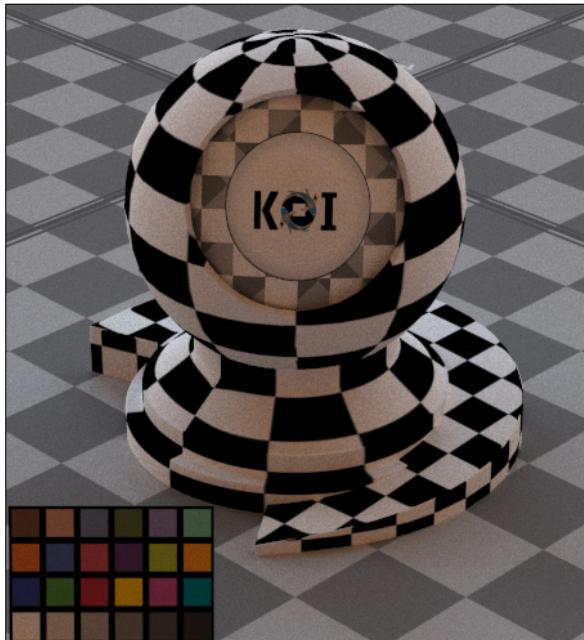
The *blend* material allows two materials to be mixed. This material is useful for applying different types of materials to the same mesh based to a mask driven by a texture map.

Attribute	Type	Defaults	Description
baseMaterial	Material	None	Specifies the first material to blend.
coatMaterial	Material	None	Specifies the second material to blend.
blendAmount	float	1.0	Specifies the amount to blend between the base and coat material. A value of 0.0 gives the base material.
blendMap	Bitmap	None	Specifies the amount to blend between the base and coat material driven by the values in the bitmap texture.

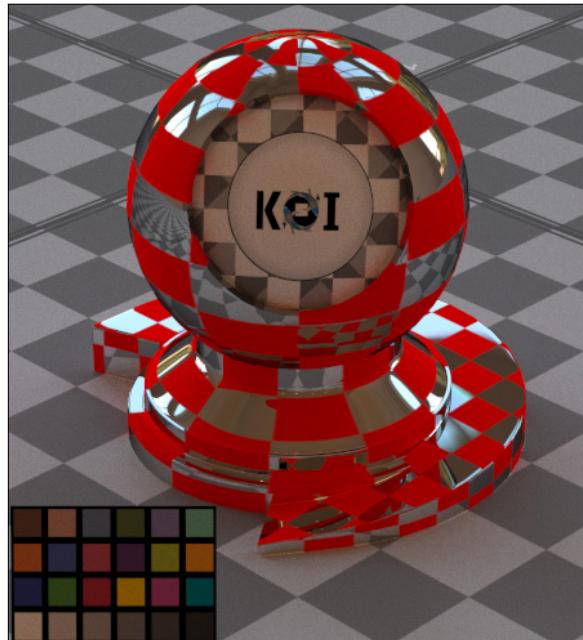
Listing 20: Defining a blend material using XML

```

1 <material type="blend">
2   <baseMaterial type="diffuse">
3     <colour value="0 0 0"/>
4   </baseMaterial>
5   <coatMaterial type="reflective">
6     <colour value="1 1 1"/>
7     <iior value="16" />
8   </coatMaterial>
9   <blendAmount value="1"/>
10  <blendMap bitmap="filepath"/>
11 </material>
```



(a) A checker texture acting as a blend mask



(b) Blend between red diffuse and reflective material mixed according to a blend mask

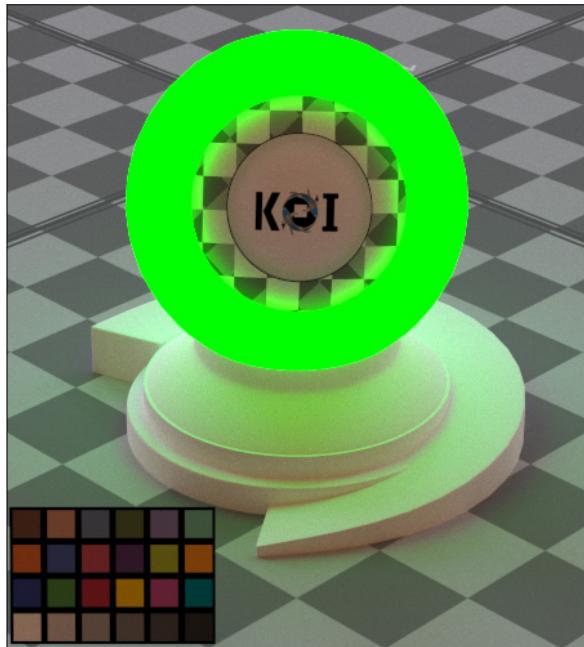
10.9. Emissive

The *emissive* material allows a surface to emit light. This material is useful for neon lights, and any other luminaire.

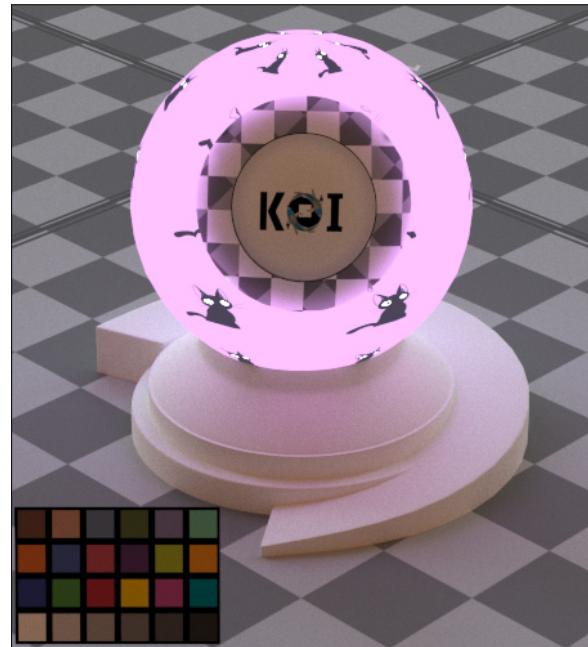
Attribute	Type	Defaults	Description
albedo	Color/Bitmap	white	Specifies the colour the material emits at any given point on the surface.
multiplier	float	1.0	Defines a scaling factor to vary the intensity.

Listing 21: Defining a emissive material using XML

```
1 <material type="emissive">
2   <multiplier value="1"/>
3   <albedo value="0 1 0"/>
4   <!-- or -->
5   <albedo bitmap="filepath"/>
6 </material>
```



(a) Rendering with a green emissive material.



(b) Rendering with a image driving the emissiveness.

10.10. Two-sided

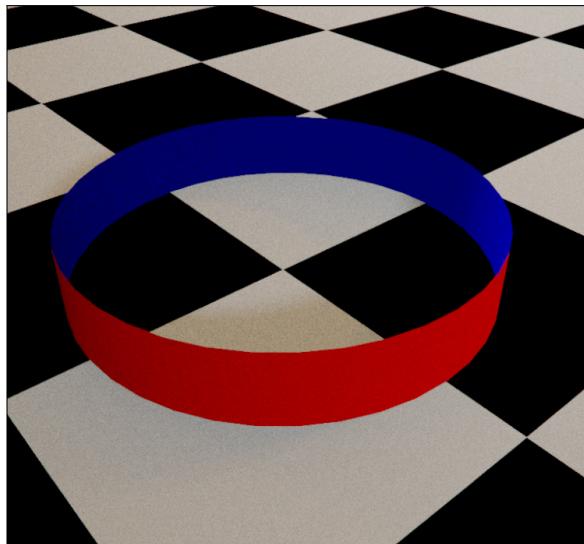
The *two sided* material allows a surface to be assigned one material if it is facing the camera, and a different material on the other side. This material is useful for open surfaces where the inside may be of a different material.

Attribute	Type	Defaults	Description
frontMaterial	Material	None	Specifies the material to assign to the front faces.
backMaterial	Material	None	Specifies the material to assign to the back faces.
blendAmount	float	1.0	Specifies the amount to blend between the front and back material.
isTransluscent	boolean	false	Specifies if the material allows light through the back face
flipDirection	boolean	false	Flips the direction the light travels through.
blendMap	Bitmap	None	Defines the location of a image to drive the translucency

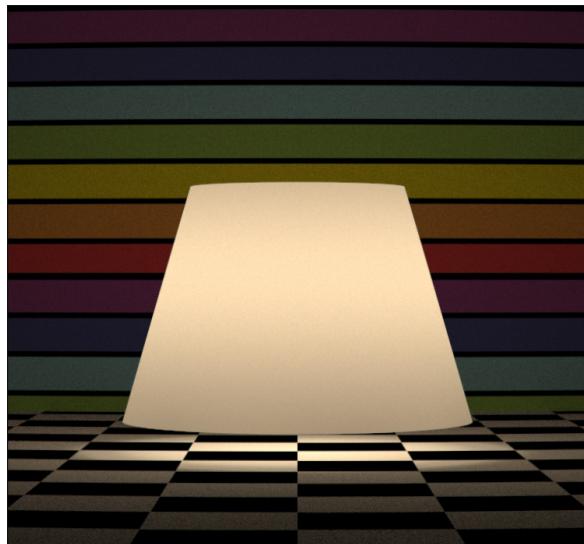
Listing 22: Defining a two-sided material using XML

```

1 <material type="twosided">
2   <frontMaterial type="diffuse">
3     <colour value="1 0 0"/>
4   </frontMaterial>
5   <backMaterial type="diffuse">
6     <albedo bitmap="filepath" />
7   </backMaterial>
8 </material>
```



(a) front:red diffuse, back:blue diffuse



(b) Rendering demonstrating translucency (the light is inside the lamp diffuser)

11. Lights

Koi uses lights to illuminate objects, without any defined in a scene nothing will be visible and only a black rendering will be generated.

There are three types of lights sources, delta, area, and infinite. Delta light sources are not physically correct because they are infinitely small and therefore have no area, and thus cannot be seen, but they are useful and easy to use. An area light source has a finite area and is visible to the renderer. Infinite light sources are infinitely far away, therefore no object in the scene can be more distant.

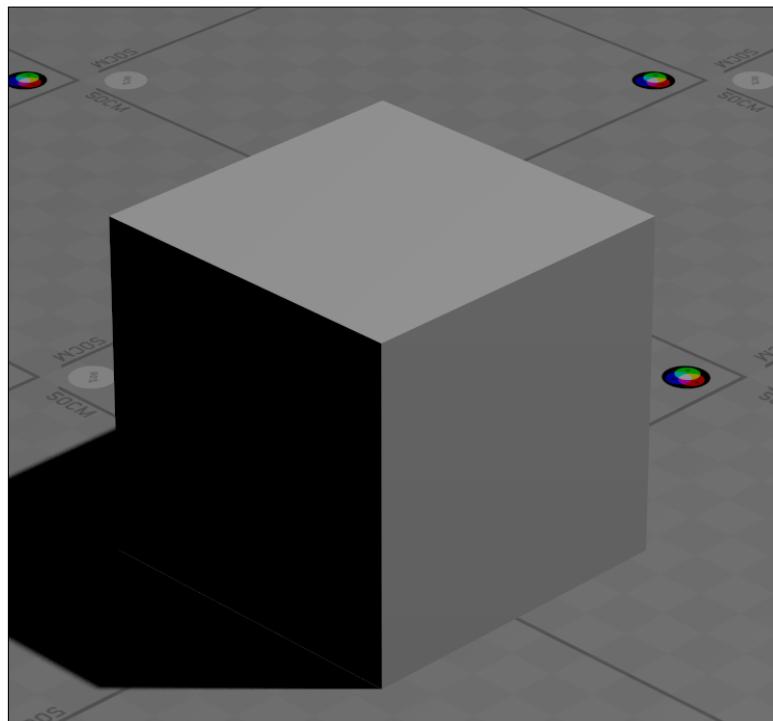
11.1. Point

A *point* light is a delta light source characterised by its position and intensity. It emits light uniformly in all directions.

Attribute	Type	Default value	Description
multiplier	float	5000.0	Defines a scaling factor to vary the intensity.
albedo	Colour	Colour(1, 1, 1)	Defines the colour of the point light.
transform	Transform	Identity transform	Defines the position of the point light.

Listing 23: Defining a point light source using XML

```
1 <light type="point">
2   <multiplier value="1000.0"/>
3   <!-- defines a red colour for the light -->
4   <albedo value="1 0 0"/>
5   <transform>
6     <max translation="0 80 100" rotation="-90 0 0"/>
7   </transform>
8 </light>
```



(a) Rendering using a point light source

11.2. Spot

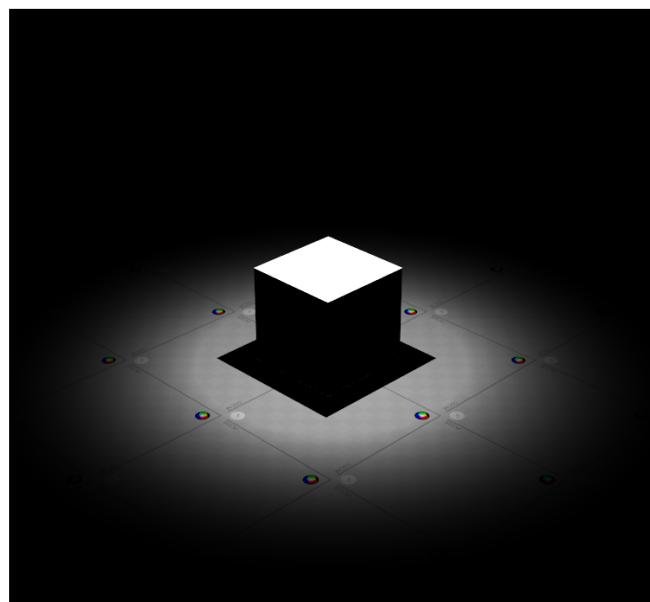
A *spot* light is a delta light source characterized by its position, rotation, tightness, and two angles called the *hotspot* and *falloff*. The default orientation of the spot light points downwards, this must be kept in mind when applying rotations.

Attribute	Type	Default value	Description
albedo	Colour	Colour(5000)	Defines the colour of the spot light.
hotspot	float	20.0	The angle in degrees defining angle of the light's beam.
falloff	float	60.0	The angle in degrees defining the outer edge. This value must be equal or greater than the hotspot angle
tightness	float	0.0	The falloff rate of the spotlight. A smaller value gives a sharper edge.
transform	Transform	Identity transform	Defines the position and rotation of the spot light.

Listing 24: Defining a spot light source using XML

```

1 <light type="spot">
2   <albedo value="2000 2000 2000"/>
3   <hotspot value="20.0"/>
4   <falloff value="60.0"/>
5   <tightness value="4"/>
6   <transform>
7     <max translation="0 80 100" rotation="0 0 0"/>
8   </transform>
9 </light>
```



(a) Rendering using a spot light source

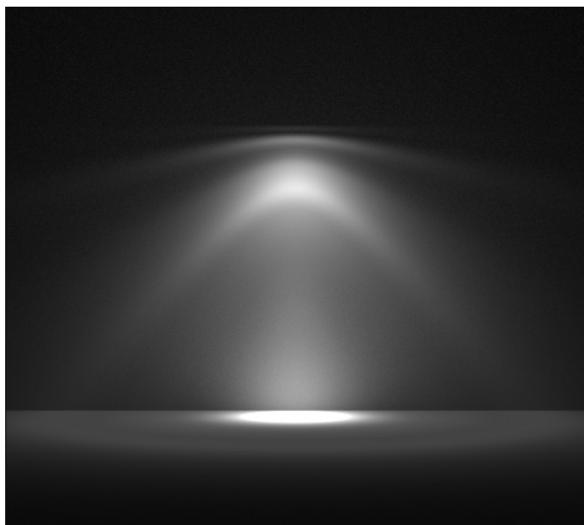
11.3. IES

An *IES* light is a delta light source using measured data from light manufacturers stored in the *.ies* photometric file format. This file format defines the distribution of light intensity at different horizontally and vertically angles. IES lights are useful for creating realistic world lighting from fixtures, or complex light distributions.

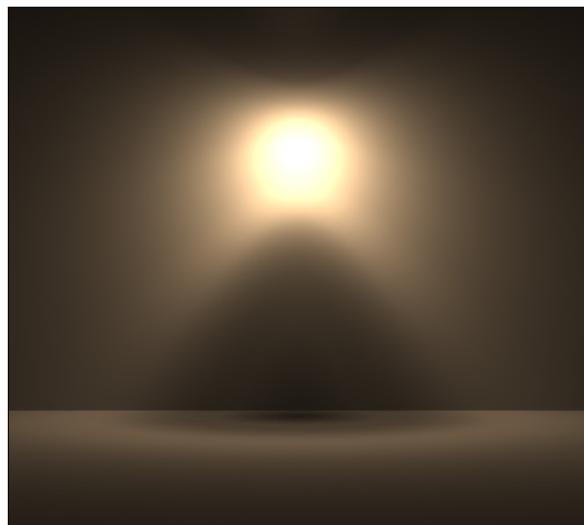
Attribute	Type	Default value	Description
file	IES	None	Specifies the location of an <i>.ies</i> file from disk.
multiplier	float	1.0	Allows the intensity of the light to be modified.
albedo	float	60.0	Defines the colour of the light.
transform	Transform	Identity transform	Defines the position and rotation of the light.

Listing 25: Defining a IES light source using XML

```
1 <light type="ies">
2   <file value="filepath"/>
3   <multiplier value="30.0"/>
4   <albedo value="1 1 1"/>
5   <transform>
6     <max translation="0 80 100" rotation="-90 0 0"/>
7   </transform>
8 </light>
```



(a) An IES light with the default colour



(b) An IES light with the albedo specified

11.4. Area

An *area* light is a physically correct light source, they have a finite area, therefore they are visible to the observer. Light is emitted uniformly over the surface area and this type of light requires a geometric object. These lights are preferred over delta light sources.

Attribute	Type	Default value	Description
geometry	Geometry	None	The geometry that will be used as a light source.
colour	Colour	1.0	Allows the intensity of the light to be modified.

Listing 26: Defining a area light source using XML

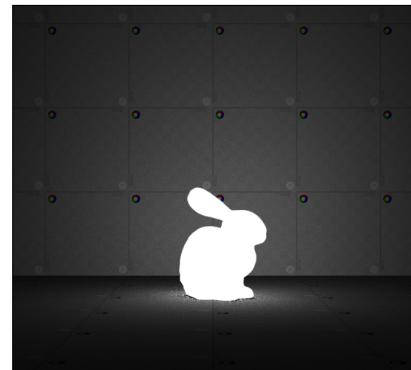
```
1 <light type="area">
2   <geometry type="quadrilateral">
3     <width value="65"/>
4     <height value="52.5"/>
5     <transform translation="278 -279.6 547.8" rotation="0 0 0"/>
6     <material type="emissive">
7       <colour value = "1 1 1"/>
8     </material>
9   </geometry>
10  <colour value="9 6.3 2.5"/>
11 </light>
```



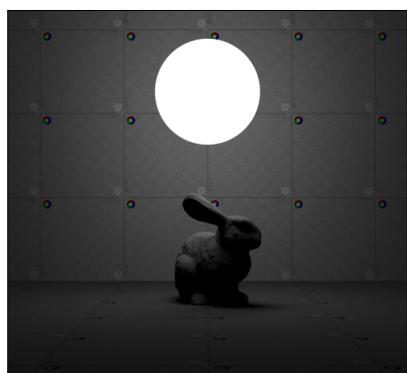
(a) Plane geometry



(b) Disc geometry



(c) Triangle mesh



(d) Sphere geometry

11.5. Dome

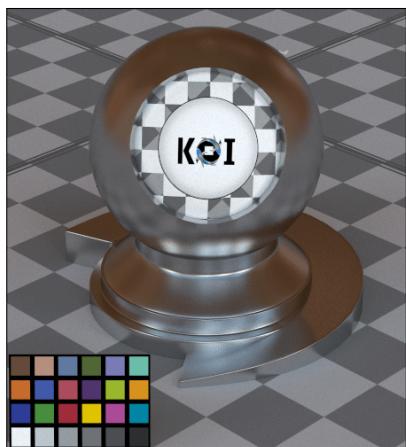
A *dome* light represents an infinite light source that wraps the scene with an HDRI image and can be used to create realistic lighting by providing both light and reflections from the environment. Dome lights are parametrised by a HDRI image, horizontal, and vertical rotations.

Attribute	Type	Default value	Description
texture	Bitmap	None	The HDRI image used.
multiplier	float	1.0	Allows the intensity of the light to be modified.
transform	Transform	Identity	Controls the horizontal (z-axis) and vertical (y-axis) rotations of the dome light. Note that translating a dome light has no effect.

Listing 27: Defining a dome light source in XML

```

1 <light type = "dome">
2   <texture pfm="filename" gamma="1.0"/>
3   <multiplier value="0.5"/>
4   <transform>
5     <max translation ="0 0 0" rotation="0 vertical horizontal"/>
6   </transform>
7 </light>
```



(a) 0° horizontal rotation



(b) 90° horizontal rotation

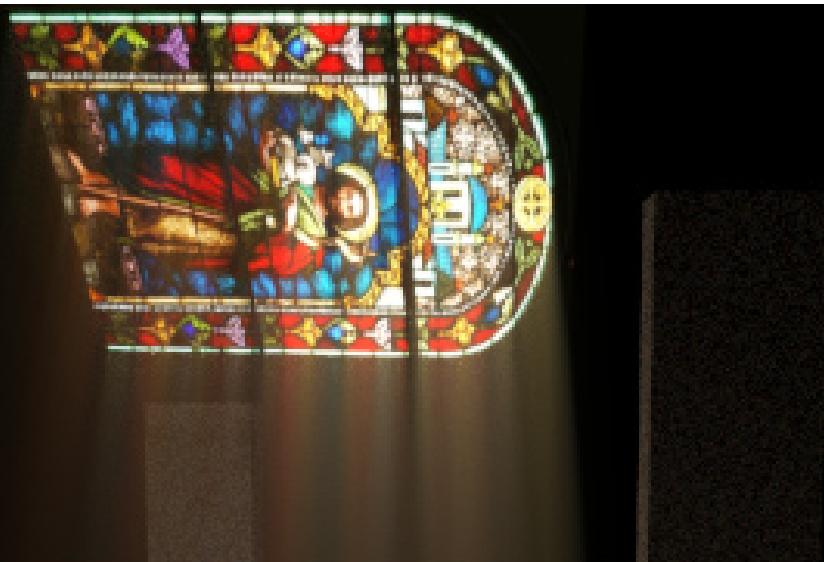


(c) 180° horizontal rotation

12. Volume

Koi supports volumetric rendering and is used to model effects such as smoke, fire, clouds, fog, steam etc.

The element ***volume*** is used in the XML description to represent them. Currently, only constant and voxel grid volume types are supported. The constant volume is isotropic meaning that the properties in the volume do not vary. In contrast the voxel grid volume can vary its properties, such as the density within its extent.



12.1. Phase

The phase function determines the overall behaviour of light scattering within a volume, they are the analogue to the BRDF for materials. There are two models supported, isotropic and henyey-greenstein phase functions. These are given by the element ***phase***.

12.1.1. Isotropic

The isotropic phase function scatters light equally in all directions

Attribute	Type	Defaults	Description
None			

Listing 28: Defining a isotropic phase function using XML

```
1 <volume type="homogeneous" stepSize="0.1">
2 ...
3   <phase type="isotropic"/>
4 ...
5 </volume>
```

12.1.2. Henyey-Greenstein

The Henyey-Greenstein phase function is an anisotropic phase function that scatters light biased towards the front, back, or equally depending on the value set by the eccentricity factor, g. A value of $g=-1.0$ will give large back-scattering, $g=0.0$ results in isotropic scattering, and $g=1.0$ results in forward-scattering.

Attribute	Type	Defaults	Description
g	float	0.0	The eccentricity value describes the bias of the direction the light travels in once it enters the medium. A positive value scatters light with a forward bias. A value of zero will scatter the light in all directions. A negative value will primarily scatter light back in the opposite direction.

Listing 29: Defining the henyey-greestein phase function using XML

```
1 <volume type="homogeneous" stepSize="0.1">
2 ...
3   <phase type="isotropic"/>
4 ...
5 </volume>
```

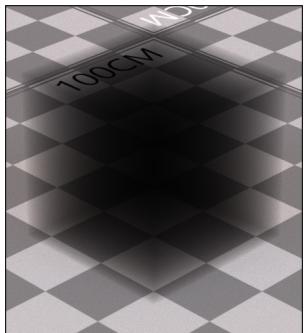
12.2. Homogeneous

The *homogeneous* volume has constant density and currently only supports a box shape as its extents.

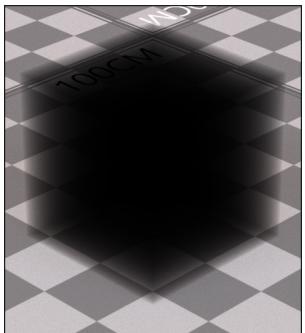
Attribute	Type	Defaults	Description
stepSize	float	1.0	The size of the step to take in centimetres to evaluate the transmittance and incoming light. A smaller value gives a more accurate result, but takes longer to render.
albedo	Colour	white	Defines the colour of the volume.
sigmaA	Colour	0.05 all channels	Defines how much of the light the volume absorbs per unit distance.
sigmaS	Colour	1.0 all channels	Defines how much of the light gets scattered per unit distance.
emission	Colour	0.0	Defines how much light the volume emits.
phase	phaseFunction	Isotropic	Defines the function that governs what fraction the light ray gets reflected in a giving direction. It is the analogue to the BRDF for materials.
bbox	BBox	min(0,0,0), max(0,0,0)	Defines the extents of the volume.

Listing 30: Defining a homogeneous volume using XML

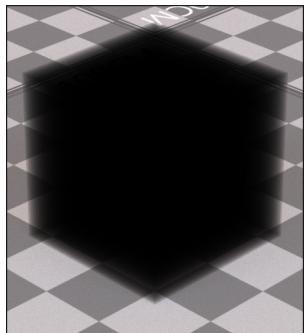
```
1 <volume type="homogeneous" stepSize="0.1">
2   <albedo value="1 0 0"/>
3   <sigmaA value="0.20 0.20 0.20"/>
4   <sigmaS value="0.00 0.00 0.00"/>
5   <emission value="0 0 0"/>
6   <phase type="hg" g="0.0" />
7   <bbox min="-25 0.01 -25" max="25 50 25" />
8 </volume>
```



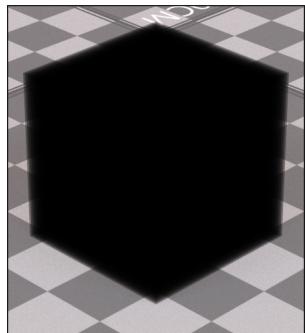
(a) sigmaA:0.1
maS:0.0



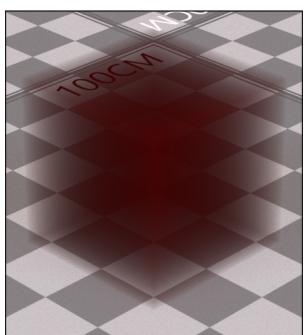
(b) sigmaA:0.2
maS:0.0



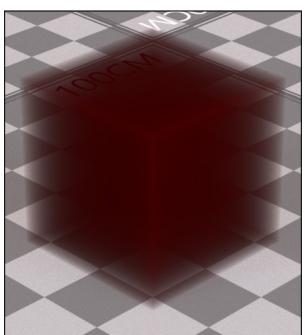
(c) sigmaA:0.4
maS:0.0



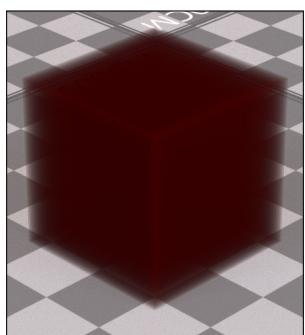
(d) sigmaA:1.0
maS:0.0



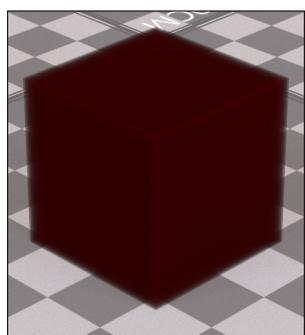
(e) sigmaA:0.0
maS:0.1



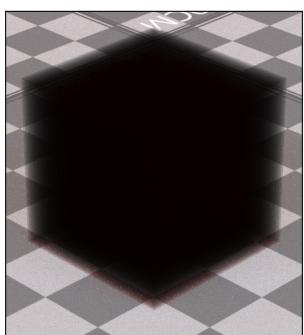
(f) sigmaA:0.0 sigmaS:0.2



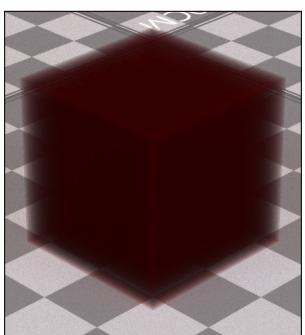
(g) sigmaA:0.0
maS:0.4



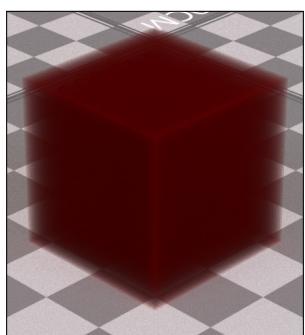
(h) sigmaA:0.0
maS:1.0



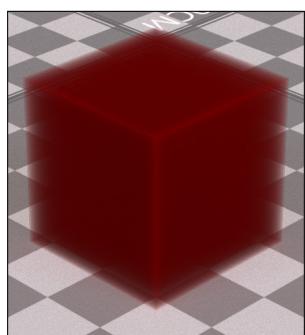
(i) g:0.99 sigmaS:0.4



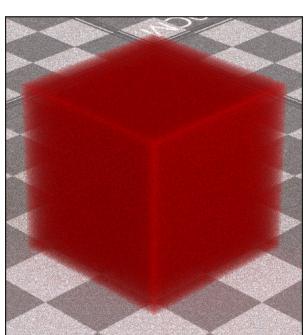
(j) g:0.75 sigmaS:0.4



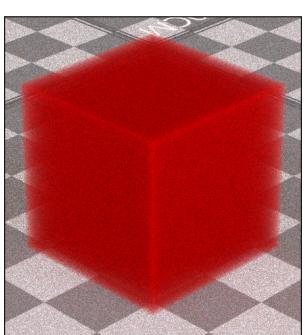
(k) g:0.50 sigmaS:0.4



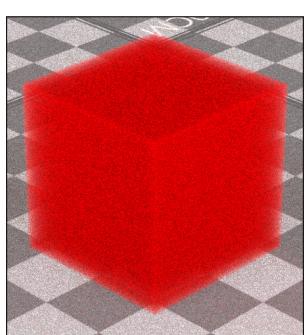
(l) g:0.25 sigmaS:0.4



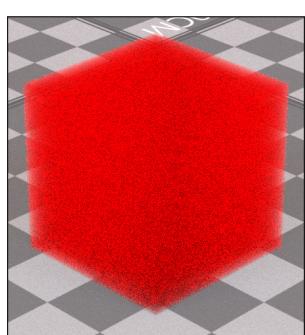
(m) g:-0.25 sigmaS:0.4



(n) g:-0.50 sigmaS:0.4



(o) g:-0.75 sigmaS:0.4



(p) g:-0.99 sigmaS:0.4

12.3. Grid

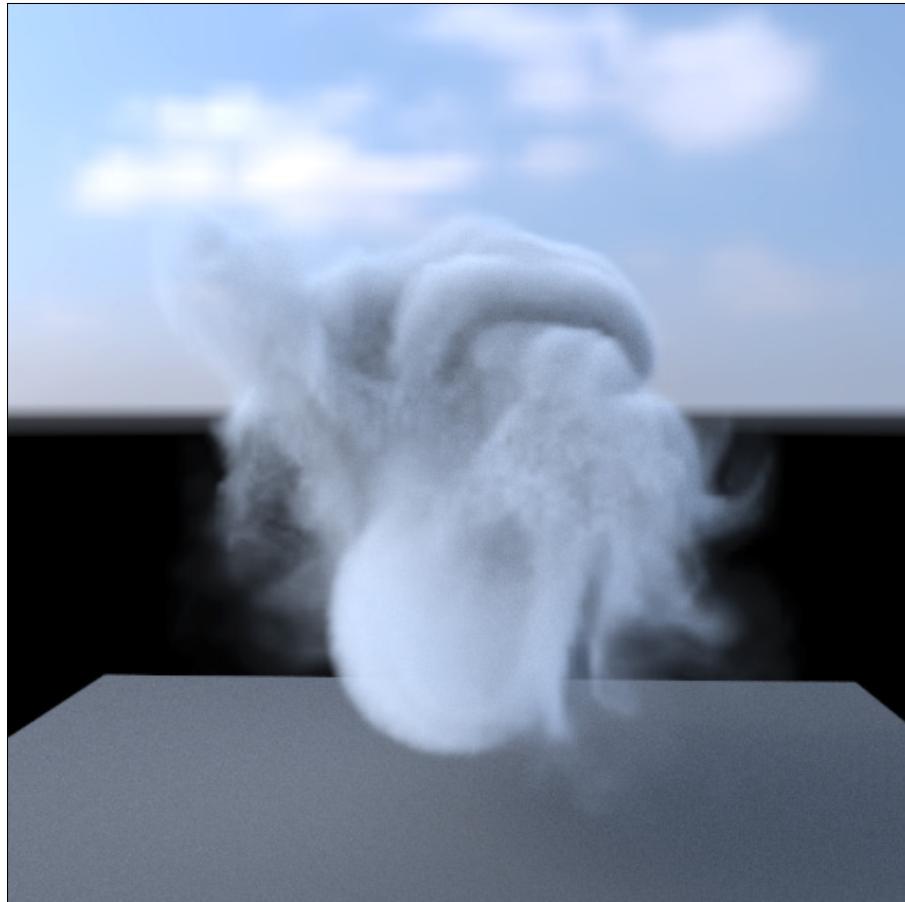
The *grid* volume has varying density and temperature. It uses a proprietary file format `*.volume` to represent a voxel buffer

Attribute	Type	Defaults	Description
stepSize	float	1.0	The size of the step to take in centimetres to evaluate the transmittance and incoming light. A smaller value gives a more accurate result, but takes longer to render.
albedo	Colour	white	Defines the colour of the volume.
sigmaA	Colour	0.05 all channels	Defines how much of the light the volume absorbs per unit distance.
sigmaS	Colour	1.0 all channels	Defines how much of the light gets scattered per unit distance.
emission	Colour	0.0	Defines how much light the volume emits.
temperature	Colour	1.0	Defines the colour of the temperature within the volume.
multiplier	float	1.0	Scales the density of the volume.
phase	phaseFunction	Isotropic	Defines the function that governs what fraction the light ray gets reflected in a giving direction. It is the analogue to the BRDF for materials.
transform	Transform	Identity	Defines the position and rotation for the volume.

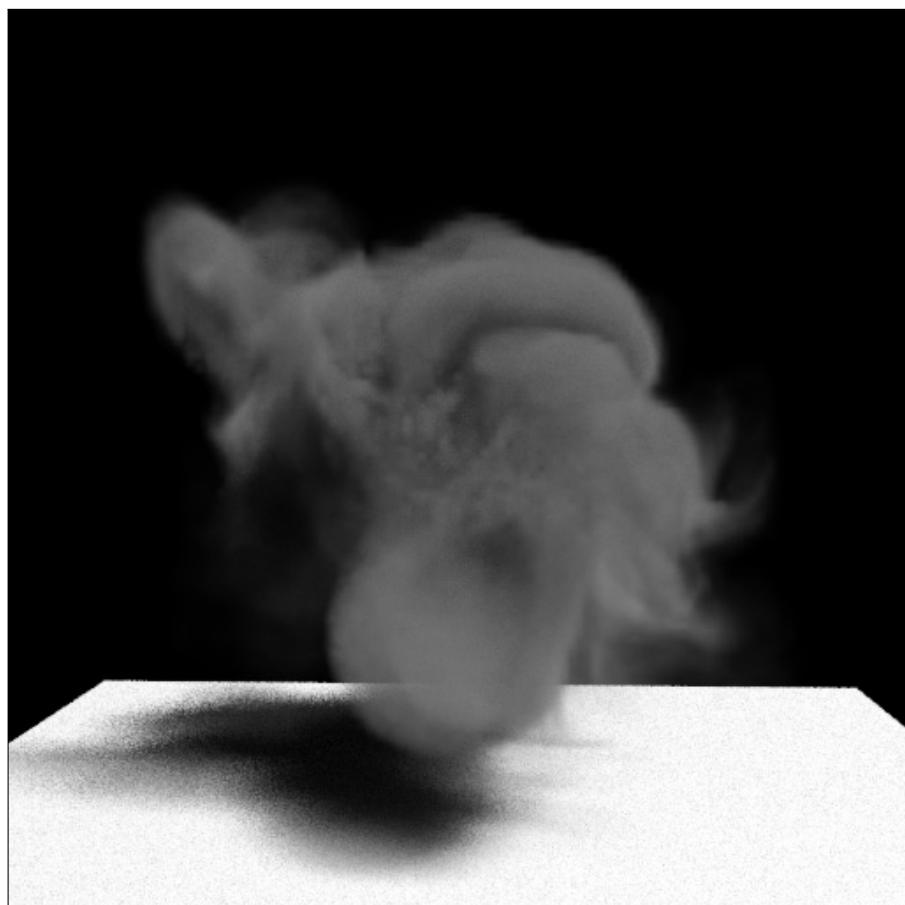
Listing 31: Defining a grid volume using XML

```

1 <volume type="grid" stepSize="0.1">
2   <file value="filepath"/>
3   <albedo value="5 0 5"/>
4   <sigmaA value="0.0 0.0 0.0"/>
5   <sigmaS value="2.00 2.00 2.00"/>
6   <emission value="0 0 0"/>
7   <temperature value="0 0 0"/>
8   <phase type="hg" g="0.0"/>
9   <multiplier value="1.127"/>
10  <transform>
11    <maya translation="0 0 0" rotation="0 0 0"/>
12  </transform>
13 </volume>
```



(a) Rendering of a volume grid with large scattering to model clouds.



(b) Rendering of a volume grid with more absorption to model smoke.

Listing 32: A python script to export a Fluid out from Maya

```
1 import os
2 import maya.cmds as cmds
3
4 filepath = 'grid.volume'
5 output = open(filepath, 'w')
6
7 sel = cmds.select("fluidShape1")
8
9 densities = cmds.getFluidAttr( at='density' )
10 temperatures = cmds.getFluidAttr( at='temperature' )
11
12 boundingBox = cmds.exactWorldBoundingBox()
13
14 resolutionW = cmds.getAttr(".resolutionW")
15 resolutionH = cmds.getAttr(".resolutionH")
16 resolutionD = cmds.getAttr(".resolutionD")
17
18 output.write(str(resolutionW) + "\n")
19 output.write(str(resolutionH) + "\n")
20 output.write(str(resolutionD) + "\n")
21
22 for axis in boundingBox:
23     output.write(str(axis) + "\n")
24
25 output.write(str(len(densities)) + "\n")
26 output.write(str(len(temperatures)) + "\n")
27
28 for density in densities:
29     output.write(str(density) + " ")
30
31 output.write("\n")
32
33 for temperature in temperatures:
34     output.write(str(temperature) + " ")
35
36 output.close()
37
38 print "Successfully saved ", os.path.abspath(filepath)
```

13. Integrator

The integrators in Koi provide different algorithms for calculating the light transport for a given scene. Currently there are five integrators, but only the whitted, path tracing, and photon mapper, are available to use in the viewer.



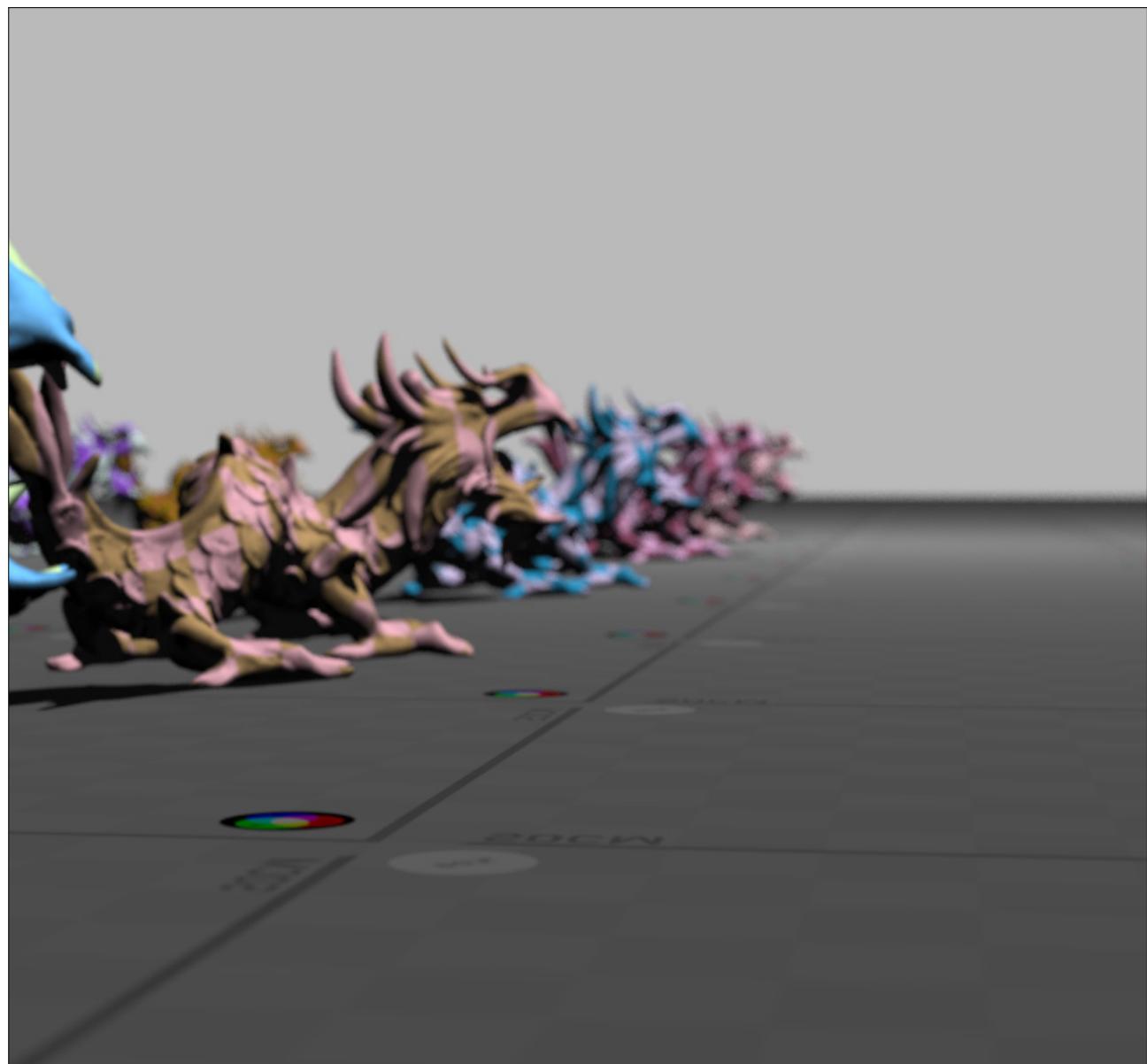
13.1. Whitted

The whitted integrator implements Whitted's recursive ray tracing algorithm.

Attribute	Type	Defaults	Description
maxBounces	int	10	The maximum number of interactions the ray can perform before it terminates.

Listing 33: Defining a whitted integrator using XML

```
1 <integrator type="whitted">
2   <maxBounces value="1"/>
3 </integrator>
```



(a) Rendering using the whitted integrator

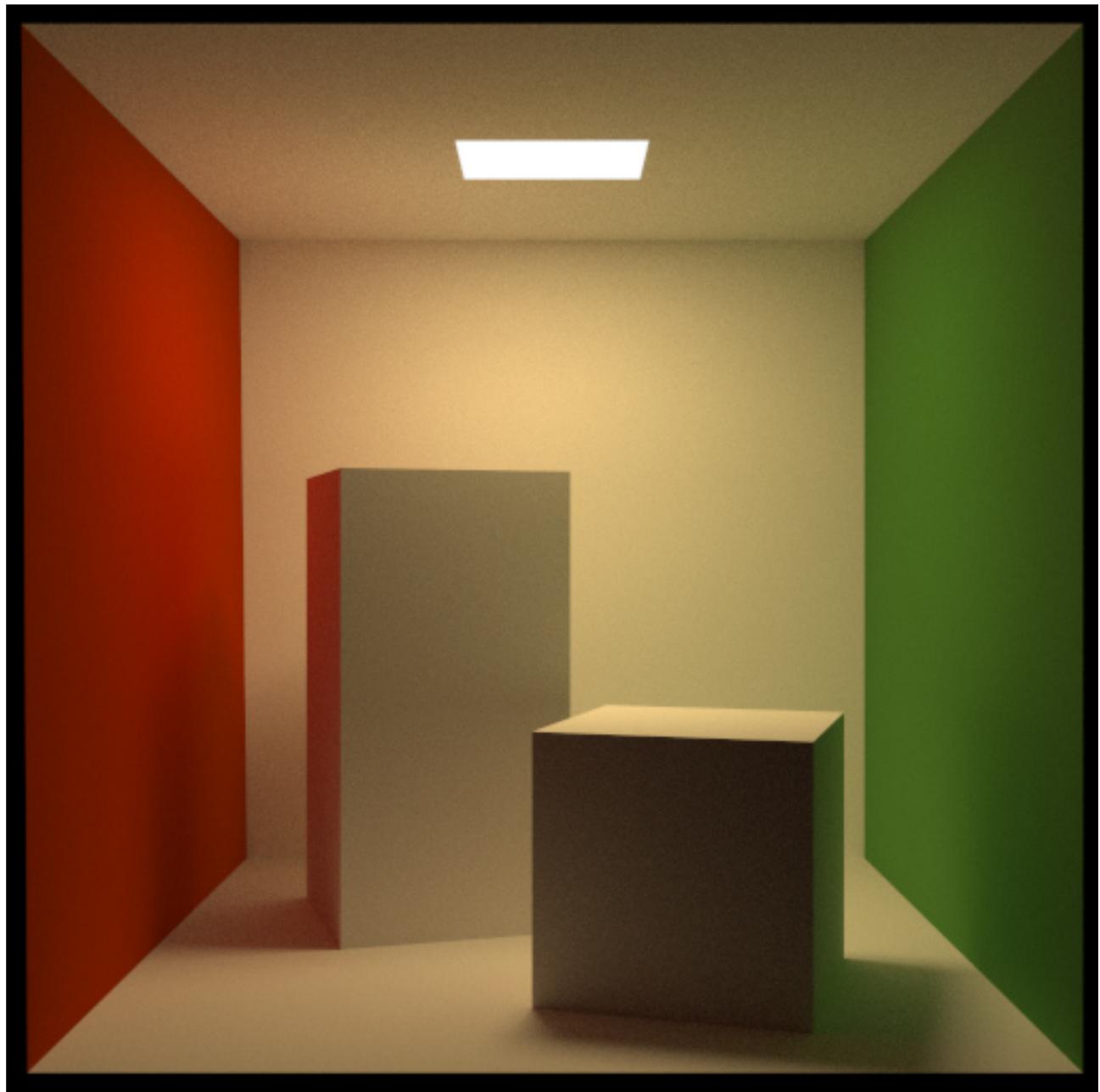
13.2. Path

The *path* integrator implements a recursive path tracing algorithm that can simulate global illumination.

Attribute	Type	Defaults	Description
maxBounces	int	10	The maximum number of interactions the ray can perform before it terminates.

Listing 34: Defining a path-tracing integrator using XML

```
1 <integrator type="path">
2   <maxBounces value="1"/>
3 </integrator>
```



(a) Rendering using the path tracing integrator

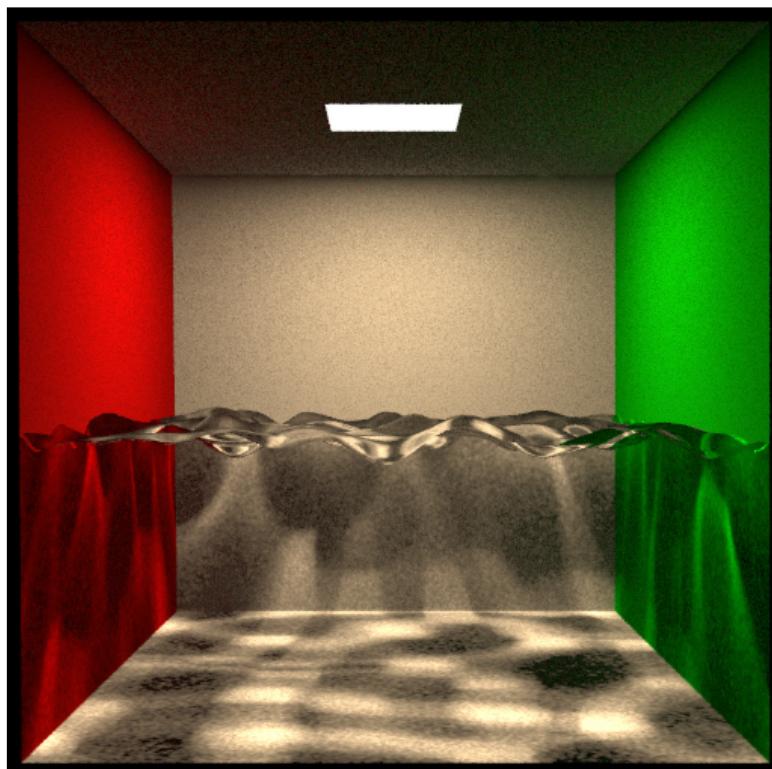
13.3. Photon

The *photon* integrator implements a photon mapping algorithm that can simulate global illumination.

Attribute	Type	Defaults	Description
maxPhotons	int	10000	The maximum number of photons to store in the <i>global</i> photon map.
maxCausticPhotons	int	0	The maximum number of photons to store in the <i>caustic</i> photon map.
maxRadius	int	20	The maximum search radius when calculating the irradiance for global illumination at a point.
maxCausticRadius	int	5	The maximum search radius when calculating the irradiance for caustics at a point.

Listing 35: Defining a photon-mapping integrator using XML

```
1 <integrator type="photon">
2   <maxPhotons value = "10000" />
3   <maxCausticPhotons value="1000000" />
4   <maxRadius value="20"/>
5   <maxCausticRadius value="20"/>
6 </integrator>
```



(a) Rendering using the photon mapping integrator

Appendix B

Source Code

Originality Avowal

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Steven Ngo

April 24, 2016

Contents

Listings	6
1 Core	10
1.1 Math	10
1.1.1 Matrix	10
1.1.2 Normal	22
1.1.3 Orthonormal Basis	27
1.1.4 Point	31
1.1.5 Point2D	36
1.1.6 Statistics	39
1.1.7 Transform	42
1.1.8 Vector	46
1.2 Utility	53
1.2.1 MeshLoader	53
1.2.2 Misc	56
1.2.3 RNG	60
1.2.4 Timer	61
2 Engine	62
2.1 Abstract/Core Classes	62
2.1.1 BBox	62
2.1.2 Bitmap	69
2.1.3 BRDF	77
2.1.4 Camera	80
2.1.5 Colour	82
2.1.6 Geometry	88
2.1.7 Integrator	89
2.1.8 Intersection	90
2.1.9 Mapping	91
2.1.10 Material	92
2.1.11 Photon	94
2.1.12 Primitive	101
2.1.13 Ray	102
2.1.14 Renderer	104
2.1.15 Sample	105
2.1.16 Sampler	111
2.1.17 Scene	115

2.1.18	Texture	118
2.2	Accelerators	119
2.2.1	BVH	119
2.3	BRDF	125
2.3.1	BlinnMicrofacetBRDF	125
2.3.2	Conductor	128
2.3.3	Dielectric	129
2.3.4	DielectricBSDF	130
2.3.5	Fresnel	132
2.3.6	GlossySpecular	134
2.3.7	Lambertian	135
2.3.8	MicrofacetBRDF	136
2.3.9	SpecularReflection	141
2.3.10	WardBRDF	142
2.4	Camera	145
2.4.1	ApertureShape	145
2.4.2	OrthographicCamera	146
2.4.3	PinholeCamera	148
2.4.4	ThinLensCamera	150
2.5	Geometry	152
2.5.1	Box	152
2.5.2	Disc	155
2.5.3	Plane	158
2.5.4	Quadrilateral	160
2.5.5	Sphere	163
2.5.6	Triangle	164
2.5.7	TriangleMesh	168
2.6	Integrators	173
2.6.1	AVIntegrator	173
2.6.2	DirectIntegrator	174
2.6.3	DirectLightingIntegrator	176
2.6.4	DirectMatIntegrator	178
2.6.5	FunctionIntegrator	180
2.6.6	NormalIntegrator	181
2.6.7	PathTracingIntegrator	182
2.6.8	PhotonMappingIntegrator	187
2.6.9	WhittedIntegrator	199
2.7	Light	204
2.7.1	AreaLight	204
2.7.2	DiffuseAreaLight	205
2.7.3	DirectedAreaLight	208
2.7.4	DistantDirectionalLight	211
2.7.5	DistantDiscLight	213
2.7.6	Dome	216

2.7.7	IES	219
2.7.8	IESLight	224
2.7.9	PointLight	226
2.7.10	SpotLight	229
2.8	Materials	231
2.8.1	BlendMaterial	231
2.8.2	BlinnMaterial	233
2.8.3	DielectricMaterial	235
2.8.4	DiffuseMaterial	237
2.8.5	EmissiveMaterial	239
2.8.6	PhongMaterial	240
2.8.7	Plastic	242
2.8.8	RoughDielectricMaterial	245
2.8.9	SpecularMaterial	247
2.8.10	TwoSidedMaterial	249
2.8.11	WardMaterial	251
2.9	Primitives	253
2.9.1	GeometricPrimitive	253
2.9.2	InstancePrimitive	255
2.10	Renderers	257
2.10.1	FunctionRenderer	257
2.10.2	SimpleRenderer	259
2.10.3	TileRenderer	261
2.11	Sampler	266
2.11.1	RandomSampler	266
2.11.2	StratifiedSampler	268
2.12	Textures	271
2.12.1	BitmapTexture	271
2.12.2	CheckerTexture	274
2.12.3	DiscTexture	276
2.12.4	GradientTexture	277
2.12.5	MaskTexture	278
2.12.6	MixTexture	279
2.12.7	RadialTexture	280
2.12.8	SolidColourTexture	281
2.12.9	StripeTexture	282
2.12.10	ToneMapping	283
2.12.11	WireframeTexture	284
2.13	Volume	286
2.13.1	ConstantVolume	286
2.13.2	PhaseFunction	288
2.13.3	RayMarcher	290
2.13.4	Volume	294
2.13.5	VolumeData	296

2.13.6	VolumeIntegrator	298
2.13.7	VoxelBuffer	299
2.13.8	VoxelGrid	300
2.14	Misc	305
2.14.1	Globals	305
2.14.2	Main	306
3	Viewer	308
3.0.3	EmptyRenderer	308
3.0.4	Global	310
3.0.5	Main	312
3.0.6	RenderThread	314
3.0.7	Viewer	318
3.0.8	XMLParser	323

Listings

1.1	Matrix header file	10
1.2	Matrix source file	12
1.3	Normal header file	22
1.4	Normal source file	25
1.5	Orthnormal basis header file	27
1.6	Point header file	31
1.7	Point source file	34
1.8	Point2D header file	36
1.9	Point2D source file	38
1.10	Statistics header file	39
1.11	Transform header file	42
1.12	Transform source file	43
1.13	Vector header file	46
1.14	Vector source file	49
1.15	MeshLoader header file	53
1.16	Misc header file	56
1.17	Misc header file	60
1.18	Misc header file	61
2.1	BBox header file	62
2.2	Bitmap header file	69
2.3	Bitmap source file	71
2.4	BRDF header file	77
2.5	Camera header file	80
2.6	Camera source file	81
2.7	Colour header file	82
2.8	Colour source file	84
2.9	Geometry header file	88
2.10	Integrator header file	89
2.11	Intersection header file	90
2.12	Mapping header file	91
2.13	Material header file	92
2.14	Photon header file	94
2.15	Primitive header file	101
2.16	Ray header file	102
2.17	Ray source file	103
2.18	Renderer header file	104

2.19	Sample header file	105
2.20	Sampler header file	111
2.21	Scene header file	115
2.22	Scene source file	117
2.23	Texture header file	118
2.24	BVH header file	119
2.25	BlinnMicrofacetBRDF header file	125
2.26	Conductor header file	128
2.27	Dielectric header file	129
2.28	DielectricBSDF header file	130
2.29	Fresnel header file	132
2.30	GlossySpecular header file	134
2.31	Lambertian header file	135
2.32	MicrofacetBRDF header file	136
2.33	SpecularReflection header file	141
2.34	WardBRDF header file	142
2.35	ApertureShape header file	145
2.36	OrthographicCamera header file	146
2.37	OrthographicCamera source file	147
2.38	PinholeCamera header file	148
2.39	PinholeCamera source file	149
2.40	ThinLensCamera header file	150
2.41	Box header file	152
2.42	Box source file	153
2.43	Disc header file	155
2.44	Plane header file	158
2.45	Plane source file	159
2.46	Quadrilateral header file	160
2.47	Sphere header file	163
2.48	Triangle header file	164
2.49	Triangle source file	165
2.50	TriangleMesh header file	168
2.51	TriangleMesh source file	170
2.52	AVIntegrator header file	173
2.53	DirectIntegrator header file	174
2.54	DirectLightingIntegrator header file	176
2.55	DirectMatIntegrator header file	178
2.56	FunctionIntegrator header file	180
2.57	NormalIntegrator header file	181
2.58	PathTracingIntegrator header file	182
2.59	PhotonMappingIntegrator header file	187
2.60	WhittedIntegrator header file	199
2.61	AreaLight header file	204
2.62	DiffuseAreaLight header file	205

2.63	DirectedAreaLight header file	208
2.64	DistantDirectionalLight header file	211
2.65	DistantDiscLight header file	213
2.66	Dome header file	216
2.67	IES header file	219
2.68	IESLight header file	224
2.69	PointLight header file	226
2.70	PointLight source file	227
2.71	SpotLight header file	229
2.72	BlendMaterial header file	231
2.73	BlinnMaterial header file	233
2.74	DielectricMaterial header file	235
2.75	DiffuseMaterial header file	237
2.76	EmissiveMaterial header file	239
2.77	PhongMaterial header file	240
2.78	Plastic header file	242
2.79	RoughDielectricMaterial header file	245
2.80	SpecularMaterial header file	247
2.81	TwoSidedMaterial header file	249
2.82	WardMaterial header file	251
2.83	GeometricPrimitive header file	253
2.84	InstancePrimitive header file	255
2.85	FunctionRenderer header file	257
2.86	SimpleRenderer header file	259
2.87	TileRenderer header file	261
2.88	RandomSampler header file	266
2.89	StratifiedSampler header file	268
2.90	BitmapTexture header file	271
2.91	BitmapTexture source file	272
2.92	CheckerTexture header file	274
2.93	DiscTexture header file	276
2.94	GradientTexture header file	277
2.95	MaskTexture header file	278
2.96	MixTexture header file	279
2.97	RadialTexture header file	280
2.98	SolidColourTexture header file	281
2.99	StripeTexture header file	282
2.100	ToneMapping header file	283
2.101	WireframeTexture header file	284
2.102	WireframeTexture source file	285
2.103	ConstantVolume header file	286
2.104	PhaseFunction header file	288
2.105	RayMarcher header file	290
2.106	Volume header file	294

2.107	VolumeData header file	296
2.108	VolumeIntegrator header file	298
2.109	VoxelBuffer header file	299
2.110	VoxelGrid header file	300
2.111	Globals header file	305
2.112	Main source file	306
3.1	MeshLoader header file	308
3.2	Global header file	310
3.3	main source file	312
3.4	RenderThread header file	314
3.5	RenderThread source file	317
3.6	viewer header file	318
3.7	viewer source file	320
3.8	XMLParser header file	323

1 Core

1.1 Math

1.1.1 Matrix

Listing 1.1: Matrix header file

```
1 #pragma once
2
3 #include <iostream>
4 #include "Vector.h"
5 #include "Point.h"
6 #include "../../Engine/Ray.h"
7 /*
8     Column major
9     | 0 4 8 12 |
10    | 1 5 9 13 | == [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
11    | 2 6 10 14 |
12    | 3 7 11 15 |
13 */
14 class Matrix
15 {
16 public:
17     Matrix();
18     Matrix(float, float, float, float,
19             float, float, float, float,
20             float, float, float, float,
21             float, float, float, float);
22     Matrix(float*);
23     Matrix(const Vector& a, float x, const Vector& b, float y, const
24             Vector& c, float z);
25     Matrix(const Vector& a, const Vector& b, const Vector& c);
26     //Matrix(const vec4& a, const vec4& b, const vec4& c);
27     friend
28         std::ostream& operator<<(std::ostream& os, const Matrix& m);
29 //    Matrix(Matrix&& a);
30 //    Matrix operator=(Matrix&& a);
31
32     static Matrix identity();
33     Matrix transpose();
34
35     float& operator[](int n);
36     float Matrix::operator[](int n) const;
```

```

37     Matrix operator+(const Matrix& b) const;
38     Matrix& operator+=(const Matrix& b);
39
40     Matrix operator-(const Matrix& b) const;
41     Matrix& operator-=(const Matrix& b);
42
43
44     Matrix operator*(float scalar) const;
45     friend Matrix operator*(float scalar, const Matrix& mat);
46
47     Matrix operator*(const Matrix& b) const;
48     Vector operator*(const Vector& v) const;
49     Normal operator*(const Normal& v) const;
50     Ray operator*(const Ray& v) const;
51     Point operator*(const Point& v) const;
52
53     float* to_array() { return a; }
54     static Matrix& Rotate(Matrix& mat, Vector axis, float rotate);
55     static Matrix RotateX(float rad);
56     static Matrix RotateY(float rad);
57     static Matrix RotateZ(float rad);
58     static Matrix Translate(float x, float y, float z);
59     static Matrix Scale(float x, float y, float z);
60
61     static Matrix& Translate(Matrix& mat, Vector translate);
62     static Matrix& Scale(Matrix& mat, Vector scale);
63     static Matrix LookAt(Point pos, Point target, Vector u);
64     static Matrix Perspective(float fov, float ar, float zNear, float
       zFar);
65     static Matrix Orthographic(float fov, float ar, float distance,
       float near, float far);
66     static Matrix Transpose(Matrix& mat);
67
68     static Matrix Inverse(const Matrix& matrix);
69 private:
70     float a[16];
71 };

```

Listing 1.2: Matrix source file

```
1 #include "Matrix.h"
2 #include "Point.h"
3 #include "../Utility/Misc.h"
4
5 Matrix::Matrix()
6 {
7
8     a[0] = 1;    a[1] = 0; a[2] = 0; a[3] = 0;
9     a[4] = 0;    a[5] = 1; a[6] = 0; a[7] = 0;
10    a[8] = 0;   a[9] = 0; a[10] = 1; a[11] = 0;
11    a[12] = 0;  a[13] = 0; a[14] = 0; a[15] = 1;
12 }
13
14
15 Matrix::Matrix(
16     float z, float b, float c, float d, // row 1
17     float e, float f, float g, float h, // row 2
18     float i, float j, float k, float l, // row 3
19     float m, float n, float o, float p) // row 4
20 {
21
22     a[0] = z;    a[1] = e; a[2] = i; a[3] = m;
23     a[4] = b;    a[5] = f; a[6] = j; a[7] = n;
24     a[8] = c;    a[9] = g; a[10] = k; a[11] = o;
25     a[12] = d;   a[13] = h; a[14] = l; a[15] = p;
26 }
27
28
29 Matrix::Matrix(float* f)
30 {
31     for(int i = 0; i < 16; ++i)
32     {
33         a[i] = f[i];
34     }
35 }
36
37 Matrix::Matrix(const Vector& x, const Vector& y, const Vector& z)
38 {
39
40     a[0] = x[0];    a[1] = y[0]; a[2] = z[0];    a[3] = 0;
41     a[4] = x[1];    a[5] = y[1]; a[6] = z[1];    a[7] = 0;
42     a[8] = x[2];    a[9] = y[2]; a[10] = z[2];   a[11] = 0;
43     a[12] = 0;      a[13] = 0;  a[14] = 0;      a[15] = 1;
44 }
45
46
47 Matrix::Matrix(const Vector& x, float aa, const Vector& y, float bb,
48                 const Vector& z, float cc)
49 {
50     a[0] = x[0];    a[1] = y[0]; a[2] = z[0];    a[3] = aa;
51     a[4] = x[1];    a[5] = y[1]; a[6] = z[1];    a[7] = bb;
```

```

51     a[8] = x[2];      a[9] = y[2];  a[10] = z[2];   a[11] = cc;
52     a[12] = 0;       a[13] = 0;    a[14] = 0;     a[15] = 1;
53 }
54
55 //Matrix::Matrix(const Vector4& x, const Vector4& y, const Vector4D& z)
56 //{
57 //  a[0] = x[0];      a[1] = y[0];  a[2] = z[0];   a[3] = 0;
58 //  a[4] = x[1];      a[5] = y[1];  a[6] = z[1];   a[7] = 0;
59 //  a[8] = x[2];      a[9] = y[2];  a[10] = z[2];  a[11] = 0;
60 //  a[12] = x[3];     a[13] = y[3]; a[14] = z[3];  a[15] = 1;
61 //}
62
63 static Matrix identity()
64 {
65     return Matrix{};
66 }
67
68 float& Matrix::operator[](int n)
69 {
70     return a[n];
71 }
72
73 float Matrix::operator[](int n) const
74 {
75     return a[n];
76 }
77
78 Matrix Matrix::operator+(const Matrix& b) const
79 {
80     return{
81         a[0] + b[0], a[1] + b[1], a[2] + b[2], a[3] + b[3],
82         a[4] + b[4], a[5] + b[5], a[6] + b[6], a[7] + b[7],
83         a[8] + b[8], a[9] + b[9], a[10] + b[10], a[11] + b[11],
84         a[12] + b[12], a[13] + b[13], a[14] + b[14], a[15] + b[15]
85     };
86 }
87 Matrix& Matrix::operator+=(const Matrix& b)
88 {
89
90     a[0] += b[0];  a[1] += b[1];  a[2] += b[2];  a[3] += b[3];
91     a[4] += b[4];  a[5] += b[5];  a[6] += b[6];  a[7] += b[7];
92     a[8] += b[8];  a[9] += b[9];  a[10] += b[10]; a[11] += b[11];
93     a[12] += b[12]; a[13] += b[13]; a[14] += b[14]; a[15] += b[15];
94     return *this;
95 }
96
97 Matrix Matrix::operator-(const Matrix& b) const
98 {
99     return{
100        a[0] - b[0], a[1] - b[1], a[2] - b[2], a[3] - b[3],
101        a[4] - b[4], a[5] - b[5], a[6] - b[6], a[7] - b[7],
102        a[8] - b[8], a[9] - b[9], a[10] - b[10], a[11] - b[11],

```

```

103         a[12] - b[12], a[13] - b[13], a[14] - b[14], a[15] - b[15]
104     };
105 }
106 Matrix& Matrix::operator-=(const Matrix& b)
107 {
108
109     a[0] -= b[0]; a[1] -= b[1]; a[2] -= b[2]; a[3] -= b[3];
110    a[4] -= b[4]; a[5] -= b[5]; a[6] -= b[6]; a[7] -= b[7];
111    a[8] -= b[8]; a[9] -= b[9]; a[10] -= b[10]; a[11] -= b[11];
112    a[12] -= b[12]; a[13] -= b[13]; a[14] -= b[14]; a[15] -= b[15];
113    return *this;
114 }
115
116 Matrix Matrix::operator*(const Matrix& mat) const
117 {
118
119     Matrix Result;
120     for (int i = 0; i < 4; ++i)
121     {
122         for (int y = 0; y < 4; ++y)
123         {
124             float value = 0;
125             for (int x = 0; x < 4; ++x)
126             {
127                 value += a[x * 4 + i] * mat[y * 4 + x];
128             }
129             Result[y * 4 + i] = value;
130         }
131     }
132     return Result;
133 }
134
135
136 Vector Matrix::operator*(const Vector& v) const
137 {
138     float x = v.GetX() * a[0] + v.GetY() * a[4] + v.GetZ() * a[8];
139     float y = v.GetX() * a[1] + v.GetY() * a[5] + v.GetZ() * a[9];
140     float z = v.GetX() * a[2] + v.GetY() * a[6] + v.GetZ() * a[10];
141
142     return Vector(x, y, z);
143 }
144
145 Normal Matrix::operator*(const Normal& normal) const
146 {
147     Matrix MInverse = Inverse(*this);
148     return Normal(MInverse[0] * normal[0] + MInverse[1] * normal[1] +
149                   MInverse[2] * normal[2],
150                   MInverse[4] * normal[0] + MInverse[5] * normal[1] +
151                   MInverse[6] * normal[2],
152                   MInverse[8] * normal[0] + MInverse[9] * normal[1] +
153                   MInverse[10] * normal[2]).Hat();
154 }
```

```

152     //float x = v.GetX() * a[0] + v.GetY() * a[4] + v.GetZ() * a[8];
153     //float y = v.GetX() * a[1] + v.GetY() * a[5] + v.GetZ() * a[9];
154     //float z = v.GetX() * a[2] + v.GetY() * a[6] + v.GetZ() * a[10];
155
156     //return Normal(x, y, z);
157 }
158
159 Point Matrix::operator*(const Point& v) const
160 {
161     float x = v.GetX() * a[0] + v.GetY() * a[4] + v.GetZ() * a[8] + a
162         [12];
163     float y = v.GetX() * a[1] + v.GetY() * a[5] + v.GetZ() * a[9] + a
164         [13];
165     float z = v.GetX() * a[2] + v.GetY() * a[6] + v.GetZ() * a[10] + a
166         [14];
167     float w = 1 / (v.GetX() * a[3] + v.GetY() * a[7] + v.GetZ() * a[11]
168         + a[15]);
169
170     return Point(x*w, y*w, z*w);
171 }
172
173 Ray Matrix::operator*(const Ray& ray) const
174 {
175     Vector dir = *this * ray.GetDirection();
176     Point point = *this * ray.GetOrigin();
177     Ray result(point, dir);
178     result.MinT = ray.MinT;
179     result.MaxT = ray.MaxT;
180     result.Depth = ray.Depth;
181     return result;
182 }
183
184 Matrix Matrix::transpose()
185 {
186     Matrix m;
187     for (int y = 0; y < 4; ++y)
188     {
189         for (int x = 0; x < 4; ++x)
190         {
191             m[x * 4 + y] = this->a[y * 4 + x];
192         }
193     }
194     std::ostream& operator<<(std::ostream& os, const Matrix& m)
195 {
196     for (int i = 0; i < 4; ++i)
197     {
198         for (int j = 0; j < 4; ++j)
199         {

```

```

200         char buffer[1024];
201         sprintf_s(buffer, 1024, "%8.5f ", m[j * 4 + i]);
202         // fcol major
203         //os << m[j * 4 + i] << ' ';
204         os << buffer;
205     }
206     os << '\n';
207 }
208
209     return os;
210 }
211
212
213 Matrix Matrix::LookAt(Point pos, Point target, Vector u)
214 {
215     // this is done in clip space z direction out of the screen
216     // so to look down the -z we subtract
217     Vector dir = (pos - target).Hat();
218     //Vector right = Cross(u, dir).Hat();
219     //Vector up = Cross(dir, right).Hat();
220     Vector right = Cross(dir, u).Hat();
221     Vector up = Cross(right, dir).Hat();
222
223     Matrix res(right, up, dir);
224     Matrix translate;
225     translate = Matrix::Translate(translate, Vector(-pos.GetX(), -pos.
226                                     GetY(), -pos.GetZ()));
227     return res * translate;
228 }
229
230 Matrix Matrix::Perspective(float fov, float ar, float near, float far)
231 {
232     Matrix res;
233     float t = tanf(DegToRad(fov * 0.5f));
234     float sx = 1 / (t * ar);
235     float sy = 1 / t;
236     float sz = (near + far) / (near - far);
237     float pz = (2 * far * near) / (near - far);
238
239     res[0] = sx;
240     res[5] = sy;
241     res[10] = sz;
242     // Vertical FOV
243     res[11] = -1;
244     // Horizontal FOV
245     //res[11] = -1/ar;
246     res[14] = pz;
247     res[15] = 0;
248
249     return res;
250 }
```

```

251 Matrix Matrix::Orthographic(float fov, float ar, float distance, float
252     near, float far)
253 {
254     Matrix res;
255     float tt = tanf(DegToRad(fov * 0.5f));
256     float pz = (2 * far * near) / (near - far);
257
258     res[0] = 1 / (tt * ar);
259     res[5] = 1 / tt;
260     res[10] = distance;
261     res[14] = -1;
262     res[11] = pz;
263     return res;
264 }
265 Matrix& Matrix::Translate(Matrix& mat, Vector translate)
266 {
267     mat[12] = translate[0];
268     mat[13] = translate[1];
269     mat[14] = translate[2];
270     return mat;
271 }
272
273 Matrix& Matrix::Scale(Matrix& mat, Vector scale)
274 {
275     mat[0] = scale[0];
276     mat[5] = scale[1];
277     mat[10] = scale[2];
278     return mat;
279 }
280
281 Matrix& Matrix::Rotate(Matrix& mat, Vector axis, float rotate)
282 {
283     Matrix x;
284     Matrix y;
285     Matrix z;
286     if (axis[0] != 0.0f)
287     {
288         x[5] = axis[0] * cos(rotate);
289         x[9] = axis[0] * -sin(rotate);
290         x[6] = axis[0] * sin(rotate);
291         x[10] = axis[0] * cos(rotate);
292     }
293     else if (axis[1] != 0.0f)
294     {
295         y[0] = axis[1] * cos(rotate);
296         y[8] = axis[1] * sin(rotate);
297         y[2] = axis[1] * -sin(rotate);
298         y[10] = axis[1] * cos(rotate);
299     }
300     else if (axis[2] != 0.0f)

```

```

302     {
303         z[0] = axis[2] * cos(rotate);
304         z[4] = axis[2] * -sin(rotate);
305         z[1] = axis[2] * sin(rotate);
306         z[5] = axis[2] * cos(rotate);
307     }
308     mat = z * y * x * mat;
309     return mat;
310 }
311
312 Matrix Matrix::RotateX(float rad)
313 {
314     Matrix x;
315     x[5] = cos(rad);
316     x[9] = -sin(rad);
317     x[6] = sin(rad);
318     x[10] = cos(rad);
319     return x;
320 }
321
322 Matrix Matrix::RotateY(float rad)
323 {
324     Matrix y;
325     y[0] = cos(rad);
326     y[8] = sin(rad);
327     y[2] = -sin(rad);
328     y[10] = cos(rad);
329     return y;
330 }
331
332 Matrix Matrix::RotateZ(float rad)
333 {
334     Matrix z;
335     z[0] = cos(rad);
336     z[4] = -sin(rad);
337     z[1] = sin(rad);
338     z[5] = cos(rad);
339     return z;
340 }
341
342 Matrix Matrix::Translate(float x, float y, float z)
343 {
344     Matrix translate;
345     translate[12] = x;
346     translate[13] = y;
347     translate[14] = z;
348     return translate;
349 }
350
351 Matrix Matrix::Scale(float x, float y, float z)
352 {
353     Matrix scale;

```

```

354     scale[0] = x;
355     scale[5] = y;
356     scale[10] = z;
357     return scale;
358 }
359
360 Matrix Matrix::Transpose(Matrix& mat)
361 {
362     Matrix Result;
363     for (int row = 0; row < 4; ++row) {
364         for (int col = 0; col < 4; ++col) {
365             Result[row * 4 + col] = mat[col * 4 + row];
366         }
367     }
368     return Result;
369 }
370
371
372 Matrix Matrix::operator*(float scalar) const
373 {
374     Matrix Result;
375     for (int row = 0; row < 4; ++row) {
376         for (int col = 0; col < 4; ++col) {
377             Result[row * 4 + col] = a[row * 4 + col] * scalar;
378         }
379     }
380     return Result;
381 }
382
383 Matrix operator*(float scalar, const Matrix& mat)
384 {
385     Matrix Result;
386     for (int row = 0; row < 4; ++row) {
387         for (int col = 0; col < 4; ++col) {
388             Result[row * 4 + col] = mat[row * 4 + col] * scalar;
389         }
390     }
391     return Result;
392 }
393
394 Matrix Matrix::Inverse(const Matrix& a)
395 {
396     float S0 = a[0] * a[5] -
397             a[1] * a[4];
398     float S1 = a[0] * a[9] -
399             a[1] * a[8];
400     float S2 = a[0] * a[13] -
401             a[1] * a[12];
402
403     float S3 = a[4] * a[9] -
404             a[5] * a[8];
405     float S4 = a[4] * a[13] -

```

```

406         a[5] * a[12];
407     float S5 = a[8] * a[13] -
408         a[9] * a[12];
409
410     float C0 = a[2] * a[7] -
411         a[3] * a[6];
412     float C1 = a[2] * a[11] -
413         a[3] * a[10];
414     float C2 = a[2] * a[15] -
415         a[3] * a[14];
416
417     float C3 = a[6] * a[11] -
418         a[7] * a[10];
419     float C4 = a[6] * a[15] -
420         a[7] * a[14];
421     float C5 = a[10] * a[15] -
422         a[11] * a[14];
423
424     float det = S0 * C5 - S1 * C4 + S2 * C3 + S3 * C2 - S4 * C1 + S5 *
425         C0;
426 //std::cout << det << std::endl;
427 if (!det)
428     std::cerr << "Determinant is zero; Cannot find inverse." << std
429         ::endl;
430
431     float invDet = 1 / det;
432     Matrix adj;
433     adj[0] = (a[5] * C5 - a[9] * C4 + a[13] * C3) * invDet;
434     adj[1] = (-a[1] * C5 + a[9] * C2 - a[13] * C1) * invDet;
435     adj[2] = (a[1] * C4 - a[5] * C2 + a[13] * C0) * invDet;
436     adj[3] = (-a[1] * C3 + a[5] * C1 - a[9] * C0) * invDet;
437
438     adj[4] = (-a[4] * C5 + a[8] * C4 - a[12] * C3) * invDet;
439     adj[5] = (a[0] * C5 - a[8] * C2 + a[12] * C1) * invDet;
440     adj[6] = (-a[0] * C4 + a[4] * C2 - a[12] * C0) * invDet;
441     adj[7] = (a[0] * C3 - a[4] * C1 + a[8] * C0) * invDet;
442
443     adj[8] = (a[7] * S5 - a[11] * S4 + a[15] * S3) * invDet;
444     adj[9] = (-a[3] * S5 + a[11] * S2 - a[15] * S1) * invDet;
445     adj[10] = (+a[3] * S4 - a[7] * S2 + a[15] * S0) * invDet;
446     adj[11] = (-a[3] * S3 + a[7] * S1 - a[11] * S0) * invDet;
447
448     adj[12] = (-a[6] * S5 + a[10] * S4 - a[14] * S3) * invDet;
449     adj[13] = (a[2] * S5 - a[10] * S2 + a[14] * S1) * invDet;
450     adj[14] = (-a[2] * S4 + a[6] * S2 - a[14] * S0) * invDet;
451     adj[15] = (a[2] * S3 - a[6] * S1 + a[10] * S0) * invDet;
452
453 //Matrix Matrix::operator=(Matrix&& x)
454 //{
455 //    Matrix b;

```

```
456 // b.a = x.a;
457 // return Matrix();
458 //}
459
460
461 //Matrix::Matrix(Matrix&& x)
462 //{
463 // std::cout << "HERE";
464 // a[0] = x[0];
465 //}
466
467 //Matrix mat1 = Matrix{ 1, 4, 2, 12,
468 //5, 1, 3, 3,
469 //6, 2, 4, 61,
470 //7, 7, 5, 12 };
471 //
472 //std::cout << mat1 << std::endl;
473 //Matrix mat2 = Matrix{ 12, 1, 5, 17,
474 //31, 8, 3, 6,
475 //5, 23, 2, 4,
476 //6, 3, 35, 2 };
477 //std::cout << mat2 << std::endl;
478 //
479 //std::cout << mat1 * mat2 << std::endl;
```

1.1.2 Normal

Listing 1.3: Normal header file

```
1 #pragma once
2
3 #include <iostream>
4 #include <assert.h>
5
6 class Vector;
7
8 class Normal
9 {
10 public:
11     Normal();
12     Normal(float x, float y, float z);
13     explicit Normal(const Vector& v);
14
15     Normal& operator=(const Normal& n);
16
17     Normal operator+(const Normal& n) const;
18     Normal operator-(const Normal& n) const;
19     Normal operator+(const Vector& v) const;
20
21     Normal operator*(float s) const;
22     Normal operator/(float s) const;
23
24     Normal& operator+=(const Normal& n);
25     Normal& operator+=(const Normal& n);
26     Normal& operator*=(float s);
27     Normal& operator/=(float s);
28
29     Normal operator-() const;
30
31     Normal& Hat();
32     float Length() const;
33     float LengthSquared() const;
34
35 // access/modification via access operator
36     float operator[](int index) const;
37     float& operator[](int index);
38
39     friend float Dot(const Normal& a, const Normal& b);
40     friend float Dot(const Normal& a, const Vector& b);
41
42     friend std::ostream& operator<<(std::ostream& os, const Normal& v);
43     friend std::istream& operator>>(std::istream& is, Normal& v);
44
45     friend inline Normal Normalize(const Normal& v);
46
47     float GetX() const;
48     float GetY() const;
49     float GetZ() const;
```

```

50
51     float SetX(float nx);
52     float SetY(float ny);
53     float SetZ(float nz);
54
55 private:
56     float X, Y, Z;
57
58 };
59
60 // lhs scalar multiplication
61 inline Normal operator*(float f, const Normal& v)
62 {
63     return v * f;
64 }
65
66 inline std::ostream& operator<<(std::ostream& os, const Normal& v)
67 {
68     os << "v(" << v.X << ", " << v.Y << ", " << v.Z << ")";
69     return os;
70 }
71
72 inline std::istream& operator>>(std::istream& is, Normal& v)
73 {
74     is >> v.X >> v.Y >> v.Z;
75     return is;
76 }
77
78
79 inline Normal Normalize(const Normal& v)
80 {
81     float LengthSq = v.X * v.X + v.Y * v.Y + v.Z * v.Z;
82     //assert(LengthSq != 0, "Normalize: Length is zero");
83     float InvLength = 1 / LengthSq;
84     return Normal(v.X * InvLength, v.Y * InvLength, v.Z * InvLength);
85 }
86
87 inline float Normal::GetX() const
88 {
89     return X;
90 }
91
92 inline float Normal::GetY() const
93 {
94     return Y;
95 }
96
97 inline float Normal::GetZ() const
98 {
99     return Z;
100}
101

```

```
102 inline float Normal::SetX(float nx)
103 {
104     X = nx;
105 }
106
107 inline float Normal::SetY(float ny)
108 {
109     Y = ny;
110 }
111
112 inline float Normal::SetZ(float nz)
113 {
114     Z = nz;
115 }
```

Listing 1.4: Normal source file

```
1 #include "Normal.h"
2 #include "Vector.h"
3
4 Normal::Normal()
5     : X{0}, Y{0}, Z{0}
6 {}
7
8 Normal::Normal(float x, float y, float z)
9     : X{x}, Y{y}, Z{z}
10 {
11     //this->Hat();
12 }
13 Normal::Normal(const Vector& v)
14     : X{ v.GetX() }, Y{ v.GetY() }, Z{ v.GetZ() }
15 {}
16
17 Normal& Normal::operator=(const Normal& n)
18 {
19     X = n.X;
20     Y = n.Y;
21     Z = n.Z;
22     return *this;
23 }
24
25 Normal Normal::operator*(float s) const
26 {
27     return Normal(s*X, s*Y, s*Z);
28 }
29 Normal Normal::operator+(const Normal& n) const
30 {
31     return Normal(X + n.X, Y + n.Y, Z + n.Z);
32 }
33
34 Normal Normal::operator-() const
35 {
36     return Normal(-X, -Y, -Z);
37 }
38
39 Normal& Normal::Hat()
40 {
41     X /= Length();
42     Y /= Length();
43     Z /= Length();
44     return *this;
45 }
46 float Normal::Length() const
47 {
48     return sqrtf(X * X + Y * Y + Z * Z);
49 }
50
51 float Dot(const Normal& a, const Normal& b)
```

```
52  {
53      return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.
54          GetZ();
55
56 float Dot(const Normal& a, const Vector& b)
57 {
58     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.
59         GetZ();
60
61 Normal Normal::operator+(const Vector& v) const
62 {
63     return Normal(X + v.GetX(), Y + v.GetY(), Z + v.GetZ());
64 }
65
66 float& Normal::operator[](int index)
67 {
68     return *(&X + index);
69 }
70
71 float Normal::operator[](int index) const
72 {
73     return *(&X + index);
74 }
```

1.1.3 Orthonormal Basis

Listing 1.5: Orthnormal basis header file

```
1 #pragma once
2
3 #include "Vector.h"
4 #include "../Utility/Misc.h"
5
6 #define ORTHONORMAL_EPSILON FLT_EPSILON
7
8 class OrthonormalBasis
9 {
10 public:
11     OrthonormalBasis();
12     OrthonormalBasis(const Vector& u, const Vector& v, const Vector& w)
13         ;
14
15     static OrthonormalBasis FromU(const Vector& u);
16     static OrthonormalBasis FromV(const Vector& v);
17     static OrthonormalBasis FromW(const Vector& w);
18
19     static Vector ToLocal(const OrthonormalBasis& basis, const Vector&
20                           v);
21     static Vector ToWorld(const OrthonormalBasis& basis, const Vector&
22                           v);
23
24     // These functions Assume that the vector w is already in the local
25     // coordinate system.
26     static float CosTheta(const Vector& w);
27     static float AbsCosTheta(const Vector& w);
28
29     static float SinTheta(const Vector& w);
30     static float SinTheta2(const Vector& w);
31
32     static float CosPhi(const Vector& w);
33     static float SinPhi(const Vector& w);
34
35     bool IsValid() const;
36     bool operator==(const OrthonormalBasis& o) const;
37     bool operator!=(const OrthonormalBasis& o) const;
38
39     friend std::ostream& operator<<(std::ostream& os, const
40                                         OrthonormalBasis& o);
41
42     Vector U, V, W;
43 };
44
45 inline OrthonormalBasis::OrthonormalBasis()
46     : U(1.0f, 0.0f, 0.0f), V(0.0f, 1.0f, 0.0f), W(0.0f, 0.0f, 1.0f)
47 {}
```

```

45 inline OrthonormalBasis::OrthonormalBasis(const Vector& u, const Vector
    & v, const Vector& w)
46     : U(u), V(v), W(w)
47 {}
48
49 inline OrthonormalBasis OrthonormalBasis::FromU(const Vector& u)
50 {
51     OrthonormalBasis basis;
52     basis.U = Normalize(u);
53     Vector::OrthonormalBasis(basis.V, basis.W, basis.U);
54     return basis;
55 }
56
57 inline OrthonormalBasis OrthonormalBasis::FromV(const Vector& v)
58 {
59     OrthonormalBasis basis;
60     basis.V = Normalize(v);
61     Vector::OrthonormalBasis(basis.U, basis.W, basis.V);
62     return basis;
63 }
64
65 inline OrthonormalBasis OrthonormalBasis::FromW(const Vector& w)
66 {
67     OrthonormalBasis basis;
68     basis.W = Normalize(w);
69     Vector::OrthonormalBasis(basis.U, basis.V, basis.W);
70     return basis;
71 }
72
73 // 3.3 Orthonormal Basis http://immersivemath.com/ila/ch03_dotproduct/
    ch03.html
74 inline Vector OrthonormalBasis::ToLocal(const OrthonormalBasis& basis,
    const Vector& v)
75 {
76     // Project v onto the orthonormal basis
77     return Vector(Dot(basis.U, v), Dot(basis.V, v), Dot(basis.W, v));
78 }
79
80 inline Vector OrthonormalBasis::ToWorld(const OrthonormalBasis& basis,
    const Vector& v)
81 {
82     return v.GetX() * basis.U + v.GetY() * basis.V + v.GetZ() * basis.W
        ;
83 }
84
85 inline float OrthonormalBasis::CosTheta(const Vector& w)
86 {
87     return w.GetZ();
88 }
89
90 inline float OrthonormalBasis::AbsCosTheta(const Vector& w)
91 {

```

```

92     return fabsf(w.GetZ());
93 }
94
95 inline float OrthonormalBasis::SinTheta(const Vector& w)
96 {
97     return sqrtf(SinTheta2(w));
98 }
99
100 inline float OrthonormalBasis::SinTheta2(const Vector& w)
101 {
102     return (1.0f - CosTheta(w) * CosTheta(w));
103 }
104
105 inline float OrthonormalBasis::CosPhi(const Vector& w)
106 {
107     float sinTheta = SinTheta(w);
108     if (sinTheta == 0.0f)
109     {
110         return 1.0f;
111     }
112     float CosPhi = w.GetX() / sinTheta;
113     // Clamp to 0-360 degrees
114     return clamp(CosPhi, -1.0f, 1.0f);
115 }
116
117
118 inline float OrthonormalBasis::SinPhi(const Vector& w)
119 {
120     float sinTheta = SinTheta(w);
121     if (sinTheta == 0.0f)
122     {
123         return 0.0f;
124     }
125     float SinPhi = w.GetY() / sinTheta;
126     return clamp(SinPhi, -1.0f, 1.0f);
127 }
128
129 inline bool OrthonormalBasis::IsValid() const
130 {
131     return (fabsf(Dot(U, V)) < ORTHONORMAL_EPSILON && fabsf(Dot(V,W)) <
132             ORTHONORMAL_EPSILON && fabsf(Dot(U,W)) < ORTHONORMAL_EPSILON);
133 }
134 inline bool OrthonormalBasis::operator==(const OrthonormalBasis& o)
135     const
136 {
137     return (U == o.U && V == o.V && W == o.W);
138 }
139 inline bool OrthonormalBasis::operator!=(const OrthonormalBasis& o)
140     const

```

```
141     return !(*this == o);
142 }
143
144 // See PBRT pg.425
145 // See Mitsuba API <Frame>
146 // See Realistic ray tracing 2nd edition pg.6-7
```

1.1.4 Point

Listing 1.6: Point header file

```
1 #pragma once
2
3 #include "Vector.h"
4
5 class Point
6 {
7
8 public:
9     Point();
10    Point(float x, float y, float z);
11
12    Point& operator=(const Point& p);
13    Point operator+(const Vector& v) const;
14    Point operator+(const Point& v) const;
15
16    Vector operator-(const Point& p) const;
17    Point operator-(const Vector& v) const;
18
19    Point& operator+=(const Vector& v);
20    Point operator*(float s) const;
21    Point operator/(float s) const;
22
23    Point operator-() const;
24
25    friend float Distance(const Point& p1, const Point& p2);
26    friend float DistanceSquared(const Point& p1, const Point& p2);
27
28    friend std::ostream& operator<<(std::ostream& os, const Point& v);
29    friend std::istream& operator>>(std::istream& is, Point& p);
30
31 // access/modification via access operator
32    float operator[](int index) const;
33    float& operator[](int index);
34
35    bool operator==(const Point& p) const;
36    bool operator!=(const Point& p) const;
37
38    inline float GetX() const;
39    inline float GetY() const;
40    inline float GetZ() const;
41
42    inline void SetX(float nx);
43    inline void SetY(float ny);
44    inline void SetZ(float nz);
45
46 private:
47    float X, Y, Z;
48
49 };
```

```

50
51 inline float Distance(const Point& p1, const Point& p2)
52 {
53     float dx = p2.X - p1.X;
54     float dy = p2.Y - p1.Y;
55     float dz = p2.Z - p1.Z;
56     return sqrtf(dx * dx + dy * dy + dz * dz);
57 }
58
59 inline float DistanceSquared(const Point& p1, const Point& p2)
60 {
61     float dx = p2.X - p1.X;
62     float dy = p2.Y - p1.Y;
63     float dz = p2.Z - p1.Z;
64     return (dx * dx + dy * dy + dz * dz);
65 }
66
67 inline float Point::GetX() const
68 {
69     return X;
70 }
71
72 inline float Point::GetY() const
73 {
74     return Y;
75 }
76
77 inline float Point::GetZ() const
78 {
79     return Z;
80 }
81
82 inline void Point::SetX(float nx)
83 {
84     X = nx;
85 }
86
87 inline void Point::SetY(float ny)
88 {
89     Y = ny;
90 }
91
92 inline void Point::SetZ(float nz)
93 {
94     Z = nz;
95 }
96
97
98 inline std::ostream& operator<<(std::ostream& os, const Point& p)
99 {
100     os << "P(" << p.X << ", " << p.Y << ", " << p.Z << ")";
101     return os;

```

```
102 }
103
104 inline std::istream& operator>>(std::istream& is, Point& p)
105 {
106     is >> p.X >> p.Y >> p.Z;
107     return is;
108 }
```

Listing 1.7: Point source file

```
1 #include "Point.h"
2
3 Point::Point()
4     : X(0), Y(0), Z(0)
5 {}
6
7 Point::Point(float x, float y, float z)
8     : X(x), Y(y), Z(z)
9 {}
10
11 Point& Point::operator=(const Point& p)
12 {
13     X = p.X;
14     Y = p.Y;
15     Z = p.Z;
16     return *this;
17 }
18
19 Point Point::operator+(const Vector& v) const
20 {
21     return Point(X + v.GetX(), Y + v.GetY(), Z + v.GetZ());
22 }
23
24 // This is a limited addition operator for adding points which is an
// affine combination of points
25 // where the coefficients of the linear combination sums to 1.
26 // See Essential mathematics for games and interactive applications 2E
// Pg.72
27 Point Point::operator+(const Point& p) const
28 {
29     return Point(X + p.X, Y + p.Y, Z + p.Z);
30 }
31
32 Vector Point::operator-(const Point& p) const
33 {
34     return Vector(X - p.X, Y - p.Y, Z - p.Z);
35 }
36
37 Point Point::operator-(const Vector& v) const
38 {
39     return Point(X - v.GetX(), Y - v.GetY(), Z - v.GetZ());
40 }
41
42 Point& Point::operator+=(const Vector& v)
43 {
44     X += v.GetX();
45     Y += v.GetY();
46     Z += v.GetZ();
47     return *this;
48 }
49
```

```

50
51 float Point::operator[](int index) const
52 {
53     return *(&X + index);
54 }
55
56 float& Point::operator[](int index)
57 {
58     return *(&X + index);
59 }
60
61 bool Point::operator==(const Point& p) const
62 {
63     return X == p[0] && Y == p[1] && Z == p[2];
64 }
65
66 bool Point::operator!=(const Point& p) const
67 {
68     return !(*this == p);
69 }
70
71 Point Point::operator*(float s) const
72 {
73     return Point(X*s, Y*s, Z*s);
74 }
75
76 Point Point::operator/(float s) const
77 {
78     s = 1 / s;
79     return Point(X * s, Y * s, Z * s);
80 }
81
82 Point Point::operator-() const
83 {
84     return Point(-X, -Y, -Z);
85 }

```

1.1.5 Point2D

Listing 1.8: Point2D header file

```
1 #pragma once
2
3 // #include "Vector.h"
4 #include <iostream>
5
6 class Point2D
7 {
8
9 public:
10    Point2D();
11    Point2D(const Point2D& other)
12    {
13        X = other.X;
14        Y = other.Y;
15    }
16
17    Point2D(float x, float y);
18
19    Point2D operator+(const Point2D& v) const;
20
21    // Vector operator-(const Point2D& p) const;
22    // Point2D operator-(const Vector& v) const;
23
24    Point2D operator*(float s) const;
25    Point2D operator/(float s) const;
26
27    Point2D operator-() const;
28
29    friend float Distance(const Point2D& p1, const Point2D& p2);
30    friend float DistanceSquared(const Point2D& p1, const Point2D& p2);
31
32    friend std::ostream& operator<<(std::ostream& os, const Point2D& v)
33        ;
34    friend std::istream& operator>>(std::istream& is, Point2D& p);
35
36    // access/modification via access operator
37    float operator[](int index) const;
38    float& operator[](int index);
39
40    inline float GetX() const;
41    inline float GetY() const;
42
43    inline void SetX(float nx);
44    inline void SetY(float ny);
45
46 private:
47     float X, Y;
48 };
```

```

49
50 inline float Distance(const Point2D& p1, const Point2D& p2)
51 {
52     float dx = p2.X - p1.X;
53     float dy = p2.Y - p1.Y;
54     return sqrtf(dx * dx + dy * dy);
55 }
56
57 inline float DistanceSquared(const Point2D& p1, const Point2D& p2)
58 {
59     float dx = p2.X - p1.X;
60     float dy = p2.Y - p1.Y;
61     return (dx * dx + dy * dy);
62 }
63
64 inline float Point2D::GetX() const
65 {
66     return X;
67 }
68
69 inline float Point2D::GetY() const
70 {
71     return Y;
72 }
73
74
75 inline void Point2D::SetX(float nx)
76 {
77     X = nx;
78 }
79
80 inline void Point2D::SetY(float ny)
81 {
82     Y = ny;
83 }
84
85
86 inline std::ostream& operator<<(std::ostream& os, const Point2D& p)
87 {
88     os << "P2D(" << p.X << ", " << p.Y << ")";
89     return os;
90 }
91
92 inline std::istream& operator>>(std::istream& is, Point2D& p)
93 {
94     is >> p.X >> p.Y;
95     return is;
96 }

```

Listing 1.9: Point2D source file

```
1 #include "Point2D.h"
2
3 Point2D::Point2D()
4     : X( 0 ), Y( 0 )
5 {}
6
7 Point2D::Point2D(float x, float y)
8     : X(x), Y(y)
9 {}
10
11 Point2D Point2D::operator+(const Point2D& p) const
12 {
13     return Point2D(X + p.X, Y + p.Y);
14 }
15
16
17 float Point2D::operator[](int index) const
18 {
19     return *(&X + index);
20 }
21
22 float& Point2D::operator[](int index)
23 {
24     return *(&X + index);
25 }
26
27
28 Point2D Point2D::operator*(float s) const
29 {
30     return Point2D(X*s, Y*s);
31 }
32
33 Point2D Point2D::operator/(float s) const
34 {
35     return Point2D(X / s, Y / s);
36 }
37
38 Point2D Point2D::operator-() const
39 {
40     return Point2D(-X, -Y);
41 }
```

1.1.6 Statistics

Listing 1.10: Statistics header file

```
1 #pragma once
2
3 #include "Point2D.h"
4 #include <vector>
5 #include <algorithm>
6 #include "../../Engine/Bitmap.h"
7
8
9 class Distribution1D
10 {
11 public:
12     Distribution1D(std::vector<float> f);
13     void Sample1D(float uniform, float& x, float & pdf) const;
14     std::vector<float> PDF;
15     std::vector<float> CDF;
16 private:
17 };
18
19
20 inline Distribution1D::Distribution1D(std::vector<float> f)
21 {
22     float integral = 0;
23     for (auto fx : f)
24     {
25         integral += fx;
26         PDF.push_back(fx);
27     }
28
29     for (int i = 0; i < f.size(); ++i)
30     {
31         if (integral == 0)
32         {
33             PDF[i] = 0;
34             //integral = 1;
35         }
36         else
37         {
38             PDF[i] /= integral;
39         }
40     }
41     CDF.push_back(0);
42     for (int i = 1; i < f.size(); ++i)
43     {
44         CDF.push_back(CDF[i - 1] + PDF[i - 1]);
45     }
46     CDF.push_back(1);
47 //std::cout << "size pdf == " << PDF.size() << std::endl;
48 //std::cout << "size cdf == " << CDF.size() << std::endl;
49
```

```

50     //for (int i = 0; i < CDF.size(); ++i)
51     //{
52     //    std::cout << "CDF:" << CDF[i] << std::endl;
53     //}
54 }
55 }
56
57 inline void Distribution1D::Sample1D(float uniform, float& x, float &
58   pdf) const
59 {
60     int index = std::max(0, int(std::lower_bound(CDF.begin(), CDF.end()
61       , uniform) - CDF.begin() - 1));
62
63     // linearly interpolate to find probability at x
64     float t = (CDF[index + 1] - uniform) / (CDF[index + 1] - CDF[index
65       ]);
66     x = (1 - t) * index + t * (index + 1);
67     x /= PDF.size();
68     pdf = PDF[index];
69 }
70
71 class Distribution2D
72 {
73 public:
74     Distribution2D(std::vector<std::vector<float>>& f);
75     Distribution2D(Bitmap* bitmap);
76     void Sample2D(const Point2D& uniform2D, Point2D& uv, float& pdf)
77       const;
78     std::vector<Distribution1D> rowDistributions;
79 private:
80     Distribution1D* marginalDensities;
81 };
82
83 inline Distribution2D::Distribution2D(std::vector<std::vector<float>>&
84   f)
85 {
86     std::vector<float> columnSum;
87     for (const std::vector<float>& row: f)
88     {
89         float sum = 0;
90         Distribution1D d(row);
91         rowDistributions.push_back(d);
92         for (float column : row)
93         {
94             sum += column;
95         }
96         columnSum.push_back(sum);
97     }
98     marginalDensities = new Distribution1D(columnSum);
99 }

```

```

97
98 inline Distribution2D::Distribution2D(Bitmap* bitmap)
99 {
100     std::vector<std::vector<float>> f;
101     for (int i = 0; i < bitmap->GetHeight(); ++i)
102     {
103         std::vector<float> g;
104         for (int j = 0; j < bitmap->GetWidth(); ++j)
105         {
106             Colour c(bitmap->GetPixel(j, i));
107             //std::cout << (c.GetR() + c.GetG() + c.GetB()) << std::
108             endl;
109             g.push_back(c.GetG() + c.GetB() + c.GetR());
110         }
111         f.push_back(g);
112     }
113
114     std::vector<float> columnSum;
115     for (const std::vector<float>& row : f)
116     {
117         float sum = 0;
118         Distribution1D d(row);
119         rowDistributions.push_back(d);
120         for (float column : row)
121         {
122             sum += column;
123         }
124         columnSum.push_back(sum);
125     }
126     marginalDensities = new Distribution1D(columnSum);
127 }
128
129 inline void Distribution2D::Sample2D(const Point2D& uniform2D, Point2D&
130     uv, float& pdf) const
131 {
132     float pdfu;
133     float pdfv;
134     marginalDensities->Sample1D(uniform2D[0], uv[1], pdfu);
135     int i = clamp(int(uv[1] * rowDistributions.size()), 0, int((
136         rowDistributions.size() - 1)));
137     rowDistributions[i].Sample1D(uniform2D[1], uv[0], pdfv);
138     pdf = pdfu * pdfv;
139 }
```

1.1.7 Transform

Listing 1.11: Transform header file

```
1 #pragma once
2
3 #include "Matrix.h"
4 #include "Vector.h"
5 #include "../../Engine/BBox.h"
6 class Transform
7 {
8 public:
9     Transform();
10    Transform(const Matrix& matrix);
11    Transform(const Matrix& matrix, const Matrix& inverse);
12
13    Transform(const Transform& t);
14    Transform Inverse() const;
15
16    static Transform Translate(const Vector& to);
17    static Transform Scale(float x, float y, float z);
18    static Transform RotateX(float rad);
19    static Transform RotateY(float rad);
20    static Transform RotateZ(float rad);
21
22    const Matrix& GetTransform() const;
23    const Matrix& GetInverseTransform() const;
24
25    Transform operator*(const Transform& other) const;
26    Point operator*(const Point& point) const;
27    Vector operator*(const Vector& vector) const;
28    Normal operator*(const Normal& normal) const;
29    Ray operator*(const Ray& ray) const;
30    BBox operator*(const BBox& box) const;
31
32 private:
33     Matrix MTransform;
34     Matrix MInverse;
35 };
```

Listing 1.12: Transform source file

```
1 #include "Transform.h"
2
3 Transform::Transform()
4 {}
5
6 Transform::Transform(const Matrix& matrix)
7     : MTransform(matrix), MInverse(Matrix::Inverse(matrix))
8 {}
9
10 Transform::Transform(const Matrix& matrix, const Matrix& inverse)
11     : MTransform(matrix), MInverse(inverse)
12 {}
13
14 Transform::Transform(const Transform& t)
15     : MTransform(t.MTransform), MInverse(t.MInverse)
16 {}
17
18 Transform Transform::Inverse() const
19 {
20     return Transform(MInverse, MTransform);
21 }
22
23 Transform Transform::Translate(const Vector& to)
24 {
25     Matrix m;
26     Matrix inv;
27     return Transform(Matrix::Translate(m, to), Matrix::Translate(inv, -to));
28 }
29
30 Transform Transform::Scale(float x, float y, float z)
31 {
32     Matrix m;
33     Matrix inv;
34     return Transform(Matrix::Scale(m, Vector(x, y, z)), Matrix::Scale(
35         inv, Vector(1/x, 1/y, 1/z)));
36 }
37 Transform Transform::RotateX(float rad)
38 {
39     Matrix m;
40     Matrix::Rotate(m, Vector(1, 0, 0), rad);
41     return Transform(m, Matrix::Transpose(m));
42 }
43
44 Transform Transform::RotateY(float rad)
45 {
46     Matrix m;
47     Matrix::Rotate(m, Vector(0, 1, 0), rad);
48     return Transform(m, Matrix::Transpose(m));
49 }
```

```

50
51 Transform Transform::RotateZ(float rad)
52 {
53     Matrix m;
54     Matrix::Rotate(m, Vector(0, 0, 1), rad);
55     return Transform(m, Matrix::Transpose(m));
56 }
57
58 const Matrix& Transform::GetTransform() const
59 {
60     return MTransform;
61 }
62
63 const Matrix& Transform::GetInverseTransform() const
64 {
65     return MIverse;
66 }
67
68 Transform Transform::operator*(const Transform& other) const
69 {
70     Matrix transform = MTransform * other.MTransform;
71     Matrix inverse = other.MIverse * MIverse;
72     return Transform(transform, inverse);
73 }
74
75 Point Transform::operator*(const Point& point) const
76 {
77     return MTransform * point;
78 }
79
80 Vector Transform::operator*(const Vector& vector) const
81 {
82     return MTransform * vector;
83 }
84
85 Normal Transform::operator*(const Normal& normal) const
86 {
87     return Normal(MIverse[0] * normal[0] + MIverse[1] * normal[1] +
88                     MIverse[2] * normal[2],
89                     MIverse[4] * normal[0] + MIverse[5] * normal[1] +
90                     MIverse[6] * normal[2],
91                     MIverse[8] * normal[0] + MIverse[9] * normal[1] +
92                     MIverse[10] * normal[2]);
93 }
94
95 Ray Transform::operator*(const Ray& ray) const
96 {
97     return MTransform * ray;
98 }
99
100 // See Graphics Gems 1 - Page 548-550 "Transforming Axis-Aligned
101 // Bounding Boxes"

```

```
98 BBox Transform::operator*(const BBox& box) const
99 {
100     float a;
101     float b;
102     Point AMin = box[0];
103     Point AMax = box[1];
104     Point BMin, BMax;
105     BMin[0] = BMax[0] = MTransform[12];
106     BMin[1] = BMax[1] = MTransform[13];
107     BMin[2] = BMax[2] = MTransform[14];
108
109     for (auto row = 0; row < 3; ++row)
110     {
111         for (auto col = 0; col < 3; ++col)
112         {
113             auto column = col * 4;
114             a = MTransform[row + column] * AMin[col];
115             b = MTransform[row + column] * AMax[col];
116             BMin[row] = BMin[row] + std::fminf(a, b);
117             BMax[row] = BMax[row] + std::fmaxf(a, b);
118         }
119     }
120     return BBox(BMin, BMax);
121 }
```

1.1.8 Vector

Listing 1.13: Vector header file

```
1 #pragma once
2
3 #include <iostream>
4 #include <assert.h>
5 #include "Normal.h"
6 #include "Point.h"
7
8 class Normal;
9 class Point;
10
11 class Vector
12 {
13 public:
14     Vector();
15     Vector(float x, float y, float z);
16     explicit Vector(const Normal& n);
17     explicit Vector(const Point& n);
18
19     Vector& operator=(const Vector& v);
20
21     Vector operator+(const Vector& v) const;
22     Normal operator+(const Normal& n) const;
23     Vector operator-(const Vector& v) const;
24     Vector operator*(float s) const;
25     Vector operator/(float s) const;
26
27     Vector& operator+=(const Vector& v);
28     Vector& operator-=(const Vector& v);
29     Vector& operator*=(float s);
30     Vector& operator/=(float s);
31
32     Vector operator-() const;
33
34     bool operator==(const Vector& other) const;
35     bool operator!=(const Vector& other) const;
36
37     Vector& Hat();
38     float Length() const;
39     float LengthSquared() const;
40
41     // access/modification via access operator
42     float operator[](int index) const;
43     float& operator[](int index);
44
45     friend inline float Dot(const Vector& a, const Vector& b);
46     friend float Dot(const Vector& v, const Normal& n);
47     friend float Dot(Vector& v, const Normal& n);
48     friend inline Vector Cross(const Vector& a, const Vector& b);
49
```

```

50     static void OrthonormalBasis(Vector& u, Vector& v, const Vector& w)
51     ;
52     friend std::ostream& operator<<(std::ostream& os, const Vector& v);
53     friend std::istream& operator>>(std::istream& is, Vector& v);
54
55     friend inline Vector Normalize(const Vector& v);
56
57     inline float GetX() const;
58     inline float GetY() const;
59     inline float GetZ() const;
60
61     inline void SetX(float nx);
62     inline void SetY(float ny);
63     inline void SetZ(float nz);
64
65 private:
66     float X, Y, Z;
67
68 };
69
70 // lhs scalar multiplication
71 inline Vector operator*(float f, const Vector& v)
72 {
73     return v * f;
74 }
75
76
77 inline float Dot(const Vector&a, const Vector& b)
78 {
79     return a.X * b.X + a.Y * b.Y + a.Z * b.Z;
80 }
81
82
83 inline Vector Cross(const Vector& a, const Vector& b)
84 {
85
86     return Vector(a.Y * b.Z - b.Y * a.Z, b.X * a.Z - a.X * b.Z, a.X * b
87 .Y - a.Y * b.X);
88 }
89
90 inline std::ostream& operator<<(std::ostream& os, const Vector& v)
91 {
92     os << "V(" << v.X << ", " << v.Y << ", " << v.Z << ")";
93     return os;
94 }
95
96 inline std::istream& operator>>(std::istream& is, Vector& v)
97 {
98     is >> v.X >> v.Y >> v.Z;
99     return is;

```

```

100 }
101
102
103 inline Vector Normalize(const Vector& v)
104 {
105     float LengthSq = v.X * v.X + v.Y * v.Y + v.Z * v.Z;
106     //assert(LengthSq != 0, "Normalize: Length is zero");
107     float InvLength = 1 / LengthSq;
108     return Vector(v.X * InvLength, v.Y * InvLength, v.Z * InvLength);
109 }
110
111 inline float Vector::GetX() const
112 {
113     return X;
114 }
115
116 inline float Vector::GetY() const
117 {
118     return Y;
119 }
120
121 inline float Vector::GetZ() const
122 {
123     return Z;
124 }
125
126 inline void Vector::SetX(float nx)
127 {
128     X = nx;
129 }
130
131 inline void Vector::SetY(float ny)
132 {
133     Y = ny;
134 }
135
136 inline void Vector::SetZ(float nz)
137 {
138     Z = nz;
139 }

```

Listing 1.14: Vector source file

```
1 #include "Vector.h"
2
3 Vector::Vector()
4     : X{ 0 }, Y{ 0 }, Z{ 0 }
5 {}
6
7 Vector::Vector(float x, float y, float z)
8     : X{ x }, Y{ y }, Z{ z }
9 {}
10
11 Vector::Vector(const Normal& n)
12     : X{ n.GetX() }, Y{ n.GetY() }, Z{ n.GetZ() }
13 {}
14
15 Vector::Vector(const Point& n)
16     : X{ n.GetX() }, Y{ n.GetY() }, Z{ n.GetZ() }
17 {}
18
19 Vector& Vector::operator=(const Vector& v)
20 {
21     X = v.X;
22     Y = v.Y;
23     Z = v.Z;
24     return *this;
25 }
26
27 Vector Vector::operator+(const Vector& v) const
28 {
29     return Vector(X + v.X, Y + v.Y, Z + v.Z);
30 }
31
32 float Dot(const Vector& v, const Normal& n)
33 {
34     return v.GetX() * n.GetX() + v.GetY() * n.GetY() + v.GetZ() * n.
35         GetZ();
36 }
37 float Dot(Vector& v, const Normal& n)
38 {
39     return v.GetX() * n.GetX() + v.GetY() * n.GetY() + v.GetZ() * n.
40         GetZ();
41 }
42 Normal Vector::operator+(const Normal& n) const
43 {
44     return Normal(X + n.GetX(), Y + n.GetY(), Z + n.GetZ());
45 }
46
47 Vector Vector::operator-(const Vector& v) const
48 {
49     return Vector(X - v.X, Y - v.Y, Z - v.Z);
```

```

50 }
51
52 Vector& Vector::operator+=(const Vector& v)
53 {
54     X += v.X;
55     Y += v.Y;
56     Z += v.Z;
57     return *this;
58 }
59
60 Vector& Vector::operator-=(const Vector& v)
61 {
62     X -= v.X;
63     Y -= v.Y;
64     Z -= v.Z;
65     return *this;
66 }
67
68 Vector Vector::operator*(float s) const
69 {
70     return Vector(X*s, Y*s, Z*s);
71 }
72
73 Vector& Vector::operator*=(float s)
74 {
75     X *= s;
76     Y *= s;
77     Z *= s;
78     return *this;
79 }
80
81 Vector Vector::operator/(float s) const
82 {
83     s = 1 / s;
84     return Vector(X*s, Y*s, Z*s);
85 }
86
87 Vector& Vector::operator/=(float s)
88 {
89     s = 1 / s;
90     X *= s;
91     Y *= s;
92     Z *= s;
93     return *this;
94 }
95
96 Vector Vector::operator-() const
97 {
98     return Vector(-X, -Y, -Z);
99 }
100
101 bool Vector::operator==(const Vector& other) const

```

```

102 {
103     return (X == other.X && Y == other.Y && Z == other.Z);
104 }
105
106 bool Vector::operator!=(const Vector& other) const
107 {
108     return !(*this == other);
109 }
110
111 float& Vector::operator[](int index)
112 {
113     return *(&X + index);
114 }
115
116 void Vector::OrthonormalBasis(Vector& u, Vector& v, const Vector& w)
117 {
118     // Assuming that w is already normalized
119     if (fabsf(w.GetX()) > fabsf(w.GetZ()))
120     {
121         v = Vector(-w.GetY(), w.GetX(), 0.0f);
122     }
123     else
124     {
125         v = Vector(0.0f, -w.GetZ(), w.GetY());
126     }
127     v.Hat();
128     u = Cross(w, v);
129 }
130
131 float Vector::operator[](int index) const
132 {
133     return *(&X + index);
134 }
135
136 Vector& Vector::Hat()
137 {
138     float Lengthsq = X * X + Y * Y + Z * Z;
139     //assert(Lengthsq != 0, "Hat:Length is zero");
140     float Invlenght = 1 / sqrtf(Lengthsq);
141
142     X *= Invlenght;
143     Y *= Invlenght;
144     Z *= Invlenght;
145
146     return *this;
147 }
148
149 float Vector::Length() const
150 {
151     return sqrtf(X * X + Y * Y + Z * Z);
152 }
153

```

```
154 float Vector::LengthSquared() const
155 {
156     return X * X + Y * Y + Z * Z;
157 }
```

1.2 Utility

1.2.1 MeshLoader

Listing 1.15: MeshLoader header file

```
1 #pragma once
2
3 #include "../../Engine/Geometry/TriangleMesh.h"
4
5 #include <stdio.h>
6 #include <string>
7 #include <fstream>
8 #include <stdio.h>
9 #include <sstream>
10 #include <regex>
11 #include <algorithm>
12
13
14 inline bool LoadOBJ1(std::string filename, std::vector<Vertex>& Indices
15   , std::vector<Point>& Points, std::vector<Normal>& Normals, std::vector<Point>& UVWs)
16 {
17     std::ifstream InputFileStream(filename);
18     if (!InputFileStream)
19     {
20         std::cerr << "Failed to load file: " << filename << std::endl;
21         return false;
22     }
23
24     std::string Line;
25
26     int P0, N0, P1, N1, P2, N2;
27     int U0, U1, U2;
28
29     int NumberOfVertices = 0;
30     int NumberOfNormals = 0;
31     int NumberOfFaces = 0;
32     int NumberOfUVWs = 0;
33     float x, y, z;
34     while (!InputFileStream.eof() && !InputFileStream.fail())
35     {
36         InputFileStream >> Line;
37         if (Line == "v")
38         {
39             NumberOfVertices++;
40         }
41         else if (Line == "vn")
42         {
43             NumberOfNormals++;
44         }
```

```

45         else if (Line == "f")
46     {
47         NumberOfFaces++;
48     }
49 }
//Indices.reserve(NumberOfVertices);
//Points.reserve(NumberOfVertices);
//Normals.reserve(NumberOfNormals);
53
54     InputFileStream.clear();
55     InputFileStream.seekg(0);
56     std::cout << "Normals: " << NumberOfNormals << std::endl;
57     std::cout << "Vertices: " << NumberOfVertices << std::endl;
58     std::cout << "Faces: " << NumberOfFaces << std::endl;
59     std::string temp;
60
61     while (!InputFileStream.eof())
62     {
63         InputFileStream >> temp;
64         if (temp == "v")
65         {
66             InputFileStream >> temp;
67             x = atof(temp.c_str());
68             InputFileStream >> temp;
69             y = atof(temp.c_str());
70             InputFileStream >> temp;
71             z = atof(temp.c_str());
72             Point v(x, y, z);
73             Points.push_back(v);
74         }
75
76         else if (temp == "vt")
77         {
78             InputFileStream >> temp;
79             x = atof(temp.c_str());
80             InputFileStream >> temp;
81             y = atof(temp.c_str());
82             InputFileStream >> temp;
83             z = atof(temp.c_str());
84             Point UVW(x, y, z);
85             UVWs.push_back(UVW);
86         }
87         else if (temp == "vn")
88         {
89             InputFileStream >> temp;
90             x = atof(temp.c_str());
91             InputFileStream >> temp;
92             y = atof(temp.c_str());
93             InputFileStream >> temp;
94             z = atof(temp.c_str());
95             Normal Normal(x, y, z);
96             Normals.push_back(Normal);

```

```

97     }
98     else if (temp == "f")
99     {
100         // Vertex//Normal f 1//1 2//2 3//3
101        // Vertex/Texture/Normal f 1/1/1 2/3/5 1/5/7
102        std::getline(InputFileStream, temp);
103        std::string number;
104        std::vector<int> tokens;
105        int count = 0;
106        for (char& ch : temp)
107        {
108            if (isdigit(ch))
109                number += ch;
110            else if (ch == '/')
111            {
112                count++;
113                tokens.push_back(std::atoi(number.c_str()));
114                number = "";
115            }
116            else if (isspace(ch))
117            {
118                if (count > 1)
119                {
120                    tokens.push_back(std::atoi(number.c_str()));
121                    number = "";
122                    count = 0;
123                }
124            }
125        }
126        P0 = tokens[0];
127        U0 = tokens[1];
128        N0 = tokens[2];
129
130        P1 = tokens[3];
131        U1 = tokens[4];
132        N1 = tokens[5];
133
134        P2 = tokens[6];
135        U2 = tokens[7];
136        N2 = tokens[8];
137        Indices.push_back(Vertex(P0 - 1, N0 - 1, U0 - 1));
138        Indices.push_back(Vertex(P1 - 1, N1 - 1, U1 - 1));
139        Indices.push_back(Vertex(P2 - 1, N2 - 1, U2 - 1));
140
141    }
142
143 }
144 InputFileStream.close();
145 return true;
146 }
```

1.2.2 Misc

Listing 1.16: Misc header file

```
1 #pragma once
2
3 #include <cmath>
4 #include "../Math/Matrix.h"
5 #ifndef M_PI
6 #define M_PI 3.14159265359f
7 #endif
8 #define M_INVPI 0.31830988618f
9 #ifndef M_E
10 #define M_E 2.71828f
11 #endif
12
13 template<class T>
14 T Max(T a, T b)
15 {
16     return a >= b ? a : b;
17 }
18
19 template<class T>
20 T Min(T a, T b)
21 {
22     return a <= b ? a : b;
23 }
24
25
26 // Linear interpolation. T must support scalar multiplication.
27 // TODO: amount and (1-amount) are the wrong way round; need to fix
28 // functions that use this.
29 template<class T>
30 T mix(T a, T b, float amount)
31 {
32     return amount * a + (1 - amount) * b;
33 }
34
35 template<class T>
36 T lerp(T a, T b, float t)
37 {
38     return (1 - t) * a + (t * b);
39 }
40
41 // Clamps a value of type T between [min, max]
42 template<class T>
43 T clamp(T a, T min, T max)
44 {
45     if (a < min)
46         return min;
47     if (a > max)
48         return max;
49     return a;
```

```

49 }
50 template<class T>
51 T DegToRad(T degrees)
52 {
53     return degrees * M_PI / 180.f;
54 }
55
56 template<class T>
57 T RadToDeg(T radians)
58 {
59     return (radians * 180.f) / M_PI;
60 }
61
62
63 // Helper functions for converting from spherical to canonical
   coordinates
64 // See page 291 of Physically based rendering
65 inline Vector SphericalDirection(float theta, float phi)
66 {
67     float sintheta = sinf(theta);
68     float costheta = cosf(theta);
69     return Vector(sintheta * cosf(phi),
70                   sintheta * sinf(phi),
71                   costheta);
72 }
73
74
75 // Conversion from canonical coordinates to spherical coordinates theta
   , phi
76 // See page 291 of Physically based rendering
77 inline float SphericalTheta(const Vector& v)
78 {
79     return acosf(clamp(-v.GetY(), -1.f, 1.f));
80 }
81
82 inline float SphericalPhi(const Vector& v)
83 {
84     float p = atan2f(v.GetZ(), v.GetX());
85     return (p < 0.f) ? p + 2.f * M_PI : p;
86 }
87
88 // Helper function to reflect a Vector v about a Normal n.
89 inline Vector Reflect(const Vector& v, const Normal& n)
90 {
91     return Vector(2 * Dot(v, n) * n) - v;
92 }
93
94
95 template<typename T>
96 T BilinearInterp(const T& s, const T& t, const T& uv00, const T& uv10,
   const T& uv01, const T& uv11)
97 {

```

```

98     // interpolate across u to s
99     T a = uv00 * (T(1) - s) + uv10 * s;
100    T b = uv01 * (T(1) - s) + uv11 * s;
101    // linearly interpolate across v between a and b
102    return a * (T(1) - t) + b * t;
103 }
104
105 template<typename T>
106 T CubicInterp(float t, const T& x0, const T& x1, const T& x2, const T&
107   x3)
108 {
109     T a0 = (1 - t) * (1 - t) * (1 - t);
110     T a1 = 3 * t * (1 - t) * (1 - t);
111     T a2 = 3 * t * t * (1 - t);
112     T a3 = t * t * t;
113     return a0*x0 + a1*x1 + a2*x2 + a3*x3;
114 }
115
116 // The default 3dsmax camera Axis order: XYZ RotationAxis: Z
117 // The default 3dsmax camera points down the z-axis
118 // From Max->System
119 // 1 For both translation and rotation
120 // 1.1 Negate Y
121 // 1.2 Swap Y with Z
122 // 3. Subtract 90 deg rotation from X axis rotation
123 // 4. Swap Y and Z order in multiplication.
124 inline Matrix XYZMax(Vector& translation, Vector& rotation)
125 {
126     Matrix CameraToWorld;
127     Matrix z = Matrix::RotateZ(DegToRad(-rotation.GetY()));
128     Matrix y = Matrix::RotateY(DegToRad(rotation.GetZ()));
129     Matrix x = Matrix::RotateX(DegToRad(rotation.GetX() - 90.0f));
130     Matrix t = Matrix::Translate(translation.GetX(), translation.GetZ()
131                               , -translation.GetY());
132     // note the order is applied in reverse first x.
133     CameraToWorld = t * y * z * x;
134     return CameraToWorld;
135 }
136 // For transforming regular objects
137 inline Matrix XYZTransformMax(Vector& translation, Vector& rotation)
138 {
139     Matrix CameraToWorld;
140     Matrix z = Matrix::RotateZ(DegToRad(-rotation.GetY()));
141     Matrix y = Matrix::RotateY(DegToRad(rotation.GetZ()));
142     Matrix x = Matrix::RotateX(DegToRad(rotation.GetX()));
143     Matrix t = Matrix::Translate(translation.GetX(), translation.GetZ()
144                               , -translation.GetY());
145     // note the order is applied in reverse first x.
146     CameraToWorld = t * y * z * x;
147     return CameraToWorld;

```

```

147 }
148
149 inline Point PointMaxToXYZ(const Point& p)
150 {
151     return Point(p.GetX(), p.GetZ(), -p.GetY());
152 }
153
154 inline Matrix XYZTransformMaya(Vector& translation, Vector& rotation)
155 {
156     Matrix CameraToWorld;
157     Matrix z = Matrix::RotateZ(DegToRad(rotation.GetZ()));
158     Matrix y = Matrix::RotateY(DegToRad(rotation.GetY()));
159     Matrix x = Matrix::RotateX(DegToRad(rotation.GetX()));
160     Matrix t = Matrix::Translate(translation.GetX(), translation.GetY()
161         , translation.GetZ());
161     CameraToWorld = t * y * z * x;
162     return CameraToWorld;
163 }
164
165 inline int ContinuousToDiscrete(float x)
166 {
167     return int(std::floorf(x));
168 }
169
170 inline float DiscreteToContinuous(float x)
171 {
172     return float(x + 0.5f);
173 }
174
175 inline int Sign(float x)
176 {
177     return x < 0 ? -1 : 1;
178 }
```

1.2.3 RNG

Listing 1.17: Misc header file

```
1 #pragma once
2
3 #include <random>
4 #include <functional>
5
6 namespace Random
7 {
8     class RandomDouble
9     {
10         public:
11             RandomDouble()
12                 : RandomDouble(0, 1, 415415)
13             {}
14             RandomDouble(double low, double high)
15                 : rand(std::bind(std::uniform_real_distribution<double>(low, high), std::mt19937()))
16             {}
17
18             RandomDouble(double low, double high, int seed)
19                 : rand(std::bind(std::uniform_real_distribution<double>(low, high), std::mt19937(seed)))
20             {}
21
22         double operator()() const { return rand(); }
23     private:
24         std::function<double()> rand;
25     };
26
27     class RandomInt
28     {
29         public:
30             RandomInt(int low, int high)
31                 : rand(std::bind(std::uniform_int_distribution<int>(low, high), std::mt19937()))
32             {}
33
34             RandomInt(int low, int high, int seed)
35                 : rand(std::bind(std::uniform_int_distribution<int>(low, high), std::mt19937(seed)))
36             {}
37
38         double operator()() const { return rand(); }
39
40     private:
41         std::function<int()> rand;
42     };
43
44 }
```

1.2.4 Timer

Listing 1.18: Misc header file

```
1 #pragma once
2
3 #include <chrono>
4 #include <vector>
5
6 class Timer
7 {
8 public:
9     Timer()
10    {}
11
12     void Start()
13    {
14         StartTime = std::chrono::system_clock::now();
15    }
16
17     void Stop()
18    {
19         ElapsedTime = std::chrono::duration_cast<std::chrono::
20             microseconds>(std::chrono::system_clock::now() - StartTime)
21             .count();
22    }
23
24     float GetElapsedSeconds()
25    {
26         return (float)ElapsedTime / 1e6;
27    }
28
29     float GetElapsedMilliseconds()
30    {
31         return (float)ElapsedTime / 1e3;
32    }
33
34     float GetElapsedMicroseconds()
35    {
36         return (float)ElapsedTime;
37    }
38 private:
39     std::chrono::system_clock::time_point StartTime;
40     long long ElapsedTime;
41 };
```

2 Engine

2.1 Abstract/Core Classes

2.1.1 BBox

Listing 2.1: BBox header file

```
1 #pragma once
2 #include <cmath>
3 #include <iostream>
4 #include "../Core/Math/Point.h"
5 #include "../Core/Math/Vector.h"
6 #include "../Engine/Intersection.h"
7 #include <iterator>
8 #include <algorithm>
9
10 class BBox
11 {
12 public:
13     BBox() :Min{ INFINITY, INFINITY, INFINITY },
14             Max{ -INFINITY, -INFINITY, -INFINITY }{}
15
16     BBox(const Point& point) : Min{ point }, Max{ point }{}
17
18     BBox(const Point& Min, const Point& Max) :
19         Min{ fminf(Min.GetX(), Max.GetX()), fminf(Min.GetY(), Max.GetY()
20             ()), fminf(Min.GetZ(), Max.GetZ()) },
21         Max{ fmaxf(Min.GetX(), Max.GetX()), fmaxf(Min.GetY(), Max.GetY()
22             ()), fmaxf(Min.GetZ(), Max.GetZ()) }
23     {}
24
25     // Two BBoxes overlap iff the projections on all axes overlap.
26     // That is if the interval of each axis overlap.
27     bool IsOverlap(const BBox& box) const
28     {
29         return Min.GetX() <= box.Max.GetX() && box.Max.GetX() <= Max.
30             GetX() &&
31             Min.GetY() <= box.Max.GetY() && box.Max.GetY() <= Max.GetY
32             () &&
33             Min.GetZ() <= box.Max.GetZ() && box.Max.GetZ() <= Max.GetZ
34             ();
35     }
36 }
```

```

33 // returns true if the point is inside the box or exactly on the
34 // borders, false otherwise.
35 bool IsInsideOrAt(const Point& point) const
36 {
37     return (point.GetX() >= Min.GetX() && point.GetX() <= Max.GetX()
38         ()) &&
39         (point.GetY() >= Min.GetY() && point.GetY() <= Max.GetY()
40         ()) &&
41         (point.GetZ() >= Min.GetZ() && point.GetZ() <= Max.GetZ()
42         ());
43 }
44
45 // returns true if the point is inside the box, false otherwise.
46 bool IsInside(const Point& point) const
47 {
48     return (point.GetX() > Min.GetX() && point.GetX() < Max.GetX())
49         &&
50         (point.GetY() > Min.GetY() && point.GetY() < Max.GetY()) &&
51         (point.GetZ() > Min.GetZ() && point.GetZ() < Max.GetZ());
52 }
53
54 // expands the box with respect to its extents.
55 BBox ExpandBy(float x, float y, float z)
56 {
57     Vector v = Vector(x, y, z);
58     Point min = Min + v;
59     Point max = Max + v;
60     return BBox(min, max);
61 }
62
63 // expands the box from its extents by w.
64 BBox ExpandBy(float w)
65 {
66     return ExpandBy(w, w, w);
67 }
68
69 // returns the volume of the box.
70 float GetVolume()
71 {
72     return (Max.GetX() - Min.GetX()) * (Max.GetY() - Min.GetY()) *
73         (Max.GetZ() * Min.GetZ());
74 }
75
76 // returns the surface area of the box.
77 float GetSurfaceArea()
78 {
79     float x = Max.GetX() - Min.GetX();
80     float y = Max.GetY() - Min.GetY();
81     float z = Max.GetZ() - Min.GetZ();
82     return 2 * (x * y + x * z + z * y);
83 }

```

```

79 // returns the extents of the box.
80 Vector GetExtent() const
81 {
82     return Max - Min;
83 }
84
85 // returns the min or max point of box.
86 Point operator[](int index) const
87 {
88     return *(&Min + index);
89 }
90
91 // returns reference to the min or max point of box.
92 Point& operator[](int index)
93 {
94     return *(&Min + index);
95 }
96
97 // returns the addition of the bounding volume.
98 BBox operator+(const Point& point) const
99 {
100     BBox Result;
101     Result.Min.SetX(fminf(point.GetX(), Min.GetX()));
102     Result.Min.SetY(fminf(point.GetY(), Min.GetY()));
103     Result.Min.SetZ(fminf(point.GetZ(), Min.GetZ()));
104
105     Result.Max.SetX(fmaxf(point.GetX(), Max.GetX()));
106     Result.Max.SetY(fmaxf(point.GetY(), Max.GetY()));
107     Result.Max.SetZ(fmaxf(point.GetZ(), Max.GetZ()));
108
109     return Result;
110 }
111
112 BBox& operator+=(const Point& point)
113 {
114     Min.SetX(fminf(point.GetX(), Min.GetX()));
115     Min.SetY(fminf(point.GetY(), Min.GetY()));
116     Min.SetZ(fminf(point.GetZ(), Min.GetZ()));
117
118     Max.SetX(fmaxf(point.GetX(), Max.GetX()));
119     Max.SetY(fmaxf(point.GetY(), Max.GetY()));
120     Max.SetZ(fmaxf(point.GetZ(), Max.GetZ()));
121
122     return *this;
123 }
124
125 // returns the addition of the bounding volume.
126 BBox operator+(const BBox& box) const
127 {
128
129     BBox Result;
130     Result.Min.SetX(fminf(box.Min.GetX(), Min.GetX()));

```

```

131     Result.Min.SetY(fminf(box.Min.GetY(), Min.GetY()));
132     Result.Min.SetZ(fminf(box.Min.GetZ(), Min.GetZ()));
133
134     Result.Max.SetX(fmaxf(box.MaxGetX(), Max.GetX()));
135     Result.Max.SetY(fmaxf(box.Max.GetY(), Max.GetY()));
136     Result.Max.SetZ(fmaxf(box.Max.GetZ(), Max.GetZ()));
137
138     return Result;
139 }
140
141 BBox& operator+=(const BBox& box)
142 {
143     Min.SetX(fminf(box.Min.GetX(), Min.GetX()));
144     Min.SetY(fminf(box.Min.GetY(), Min.GetY()));
145     Min.SetZ(fminf(box.Min.GetZ(), Min.GetZ()));
146
147     Max.SetX(fmaxf(box.Max.GetX(), Max.GetX()));
148     Max.SetY(fmaxf(box.Max.GetY(), Max.GetY()));
149     Max.SetZ(fmaxf(box.Max.GetZ(), Max.GetZ()));
150
151     return *this;
152 }
153
154 int LongestAxis() const
155 {
156     int index = 0;
157     float max = -INFINITY;
158     Vector v = Max - Min;
159     for (int i = 0; i < 3; ++i)
160     {
161         if (abs(v[i]) > max)
162         {
163             index = i;
164             max = abs(v[i]);
165         }
166     }
167     return index;
168 }
169
170 bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
171                     Intersection& intersection) const
172 {
173     float TMin, TMax;
174     float TYMin, TYMax;
175     float TZMin, TZMax;
176
177     Vector Direction = ray.GetDirection();
178     Point Origin = ray.GetOrigin();
179
180     if (Direction.GetX() >= 0)
181     {
182         TMin = (Min.GetX() - Origin.GetX()) / Direction.GetX();

```

```

182         TMax = (Max.GetX() - Origin.GetX()) / Direction.GetX();
183     }
184     else
185     {
186         TMin = (Max.GetX() - Origin.GetX()) / Direction.GetX();
187         TMax = (Min.GetX() - Origin.GetX()) / Direction.GetX();
188     }
189
190     if (Direction.GetY() >= 0)
191     {
192         TYMin = (Min.GetY() - Origin.GetY()) / Direction.GetY();
193         TYMax = (Max.GetY() - Origin.GetY()) / Direction.GetY();
194     }
195     else
196     {
197         TYMin = (Max.GetY() - Origin.GetY()) / Direction.GetY();
198         TYMax = (Min.GetY() - Origin.GetY()) / Direction.GetY();
199     }
200     if ((TMin > TYMax) || (TYMin > TMax))
201         return false;
202
203     if (TYMin > TMin)
204         TMin = TYMin;
205
206     if (TYMax < TMax)
207         TMax = TYMax;
208
209     if (Direction.GetZ() >= 0)
210     {
211         TZMin = (Min.GetZ() - Origin.GetZ()) / Direction.GetZ();
212         TZMax = (Max.GetZ() - Origin.GetZ()) / Direction.GetZ();
213     }
214     else
215     {
216         TZMin = (Max.GetZ() - Origin.GetZ()) / Direction.GetZ();
217         TZMax = (Min.GetZ() - Origin.GetZ()) / Direction.GetZ();
218     }
219     if ((TMin > TZMax) || (TZMin > TMax))
220         return false;
221
222     if (TZMin > TMin)
223         TMin = TZMin;
224
225     if (TZMax < TMax)
226         TMax = TZMax;
227
228     return (TMax > epsilon);
229 }
230
231 bool IntersectRay(const Ray& ray, float& out_t0, float& out_t1)
232     const

```

```

233     {
234
235         float tNear = ray.MinT;
236         float tFar = ray.MaxT;
237
238         // calculate intersection with each slab and check intervals
239         for (int dimension = 0; dimension < 3; ++dimension)
240         {
241             float inverseDirection = 1.0f / ray.GetDirection()[dimension];
242             float t0 = (Min[dimension] - ray.GetOrigin()[dimension]) *
243                         inverseDirection;
244             float t1 = (Max[dimension] - ray.GetOrigin()[dimension]) *
245                         inverseDirection;
246             if (t0 > t1)
247             {
248                 std::swap(t0, t1);
249             }
250             tNear = std::max(tNear, t0);
251             tFar = std::min(tFar, t1);
252             if (tNear > tFar)
253             {
254                 return false;
255             }
256             if (tFar < 0.0f)
257             {
258                 return false;
259             }
260             out_t0 = tNear;
261             out_t1 = tFar;
262             return true;
263         }
264
265         // returns the center point of the box.
266         Point GetCenter() const
267     {
268         return (Min + Max) / 2;
269     }
270
271         friend std::ostream& operator<<(std::ostream& os, const BBox& box)
272     {
273         os << "bbox<" << box.Min << box.Max << ">";
274         return os;
275     }
276
277
278         friend std::istream& operator>>(std::istream& is, BBox& box)
279     {
280         is >> box.Min >> box.Max;
281         return is;

```

```
282     }
283
284     Vector RelativePosition(Point point) const;
285 private:
286     Point Min;
287     Point Max;
288 };
289
290 inline Vector BBox::RelativePosition(Point point) const
291 {
292     return Vector((point[0] - Min[0]) / (Max[0] - Min[0]),
293                  (point[1] - Min[1]) / (Max[1] - Min[1]),
294                  (point[2] - Min[2]) / (Max[2] - Min[2]));
295 }
```

2.1.2 Bitmap

Listing 2.2: Bitmap header file

```
1 #pragma once
2 // For now default format is simply RGB888 (0xffRRGGBB)
3 #include "Colour.h"
4
5 class Bitmap
6 {
7 public:
8     Bitmap(int width, int height);
9     Bitmap() {}
10    ~Bitmap();
11
12    // Move Ctor
13    Bitmap(Bitmap&& b) : Pixels{ std::move(b.Pixels) }, Width{b.Width},
14        Height{b.Height}, Gamma(b.Gamma) {
15        b.Pixels = nullptr;
16    }
17
18    // Copy ctor
19    Bitmap(const Bitmap& b)
20    {
21        Gamma = b.Gamma;
22        Width = b.Width;
23        Height = b.Height;
24        Pixels = new Colour*[b.GetHeight()];
25        for (int i = 0; i < b.GetHeight(); ++i)
26        {
27            Pixels[i] = new Colour[b.GetWidth()];
28            for (int j = 0; j < b.GetWidth(); ++j)
29                Pixels[i][j] = b.GetPixel(j, i);
30        }
31    }
32    Bitmap& operator=(const Bitmap& b)
33    {
34        if (this != &b)
35        {
36            Gamma = b.Gamma;
37            Width = b.Width;
38            Height = b.Height;
39            Pixels = new Colour*[b.GetHeight()];
40            for (int i = 0; i < b.GetHeight(); ++i)
41            {
42                Pixels[i] = new Colour[b.GetWidth()];
43                for (int j = 0; j < b.GetWidth(); ++j)
44                    Pixels[i][j] = b.GetPixel(j, i);
45            }
46        }
47        return *this;
48    }
```

```
49
50 Colour* operator[](int i) const
51 { return Pixels[i]; }
52 void SetPixel(int x, int y, Colour& c) const;
53 Colour GetPixel(int x, int y) const;
54 Colour GetLinearPixel(int x, int y) const;
55
56 void SavePPM(std::string filename) const;
57 void SavePPM6(std::string filename, float gamma = 1.f) const;
58 void SavePFM(std::string filename, float gamma = 1.f) const;
59 static Bitmap LoadPPM(std::string filename);
60 static Bitmap LoadPFM(std::string filename);
61 int GetWidth() const;
62 int GetHeight() const;
63 private:
64     int Width, Height;
65     float Gamma;
66     Colour** Pixels;
67 }
```

Listing 2.3: Bitmap source file

```
1 #include "Bitmap.h"
2 #include <fstream>
3 #include <string>
4 #include <sstream>
5 #include "../Globals.h"
6
7 Bitmap::Bitmap(int width, int height)
8     : Width{ width }, Height{ height }, Gamma{1.0f}
9 {
10     Pixels = new Colour*[height];
11     for (int i = 0; i < height; ++i)
12     {
13         Pixels[i] = new Colour[width];
14         for (int j = 0; j < width; ++j)
15             Pixels[i][j] = Colour(0.0f, 0.0f, 0.0f);
16     }
17 }
18
19 Bitmap::~Bitmap()
20 {
21     delete Pixels;
22 }
23
24 void Bitmap::SetPixel(int x, int y, Colour& c) const
25 {
26     Pixels[y][x] = c;
27 }
28
29 Colour Bitmap::GetPixel(int x, int y) const
30 {
31     return Pixels[y][x];
32 }
33
34 Colour Bitmap::GetLinearPixel(int x, int y) const
35 {
36     return Pixels[y][x] ^ (1 / 2.2);
37 }
38 void Bitmap::SavePPM(std::string filename) const
39 {
40     std::ofstream ofs(filename.c_str(), std::ios::out);
41     ofs << "P3" << std::endl;
42     ofs << "#" << filename << std::endl;
43     ofs << Width << " " << Height << std::endl;
44     ofs << 255 << std::endl;
45     Colour c;
46     int r, g, b;
47     for (int row = 0; row < Height; ++row)
48     {
49         for (int col = 0; col < Width; ++col)
50         {
51             c = Pixels[row][col];
```

```

52         r = c.GetR() * 255;
53         g = c.GetG() * 255;
54         b = c.GetB() * 255;
55         ofs << " " << r << " " << g << " " << b;
56
57     }
58     ofs << std::endl;
59 }
60
61     ofs.close();
62 #if DEBUG
63     std::cout << "Debug P3: Saved " << filename << std::endl;
64 #endif
65 }
66
67 void Bitmap::SavePPM6(std::string filename, float gamma) const
68 {
69     std::ofstream ofs(filename.c_str(), std::ios::out | std::ios::
70                     binary);
70     ofs << "P6" << std::endl;
71     ofs << Width << std::endl;
72     ofs << Height << std::endl;
73     ofs << 255 << std::endl;
74
75     Colour c;
76     unsigned char r, g, b;
77     unsigned char buffer[3];
78     for (int row = 0; row < Height; ++row)
79     {
80         for (int col = 0; col < Width; ++col)
81         {
82             c = Pixels[row][col];
83             c = c ^ (1 / gamma);
84             r = c.GetR() * 255;
85             g = c.GetG() * 255;
86             b = c.GetB() * 255;
87             ofs << r << g << b;
88         }
89     }
90     ofs.close();
91 #if DEBUG
92     std::cout << "Debug P6: Saved " << filename << std::endl;
93 #endif
94 }
95
96
97 Bitmap Bitmap::LoadPPM(std::string filename)
98 {
99     std::ifstream ifs(filename.c_str(), std::ios::binary | std::ios::in
100                );
101    if (!ifs)

```

```

102      {
103 #if DEBUG
104         std::cout << "LoadPPM: Could not open " << filename << std::
105         endl;
106     }
107     std::string format;
108     int Width, Height;
109     int MaxGray;
110     unsigned char r, g, b;
111     ifs >> format;
112     ifs >> Width >> Height;
113     ifs >> MaxGray;
114     Bitmap result(Width, Height);
115
116     std::string Line;
117     char ch;
118     unsigned char buffer[3];
119
120     while (ifs.peek() == '\n' || ifs.peek() == ' ')
121     {
122 #if DEBUG
123         std::cout << "P6: Ate some whitespace..." << std::endl;
124 #endif
125         ifs.read((char*)(&buffer[0]), 1);
126     }
127     for (int i = 0; i < Height; ++i)
128     {
129         for (int j = 0; j < Width; ++j)
130         {
131             ifs.read((char*)(&buffer[0]), sizeof(buffer));
132             int r = (int)(unsigned char)buffer[0];
133             int g = (int)(unsigned char)buffer[1];
134             int b = (int)(unsigned char)buffer[2];
135             //cout << r << " " << g << " " << b << endl;
136             result.SetPixel(j, i, Colour(r / (float)MaxGray,
137                                         g / (float)MaxGray,
138                                         b / (float)MaxGray));
139         }
140     }
141 }
142
143     ifs.close();
144     return result;
145 #if DEBUG
146     std::cout << "Debug: Loaded " << filename << std::endl;
147 #endif
148 }
149
150
151 Bitmap Bitmap::LoadPFM(std::string filename)
152 {

```

```

153     std::ifstream ifs(filename.c_str(), std::ios::binary | std::ios::in
154         );
155     if (!ifs)
156     {
157 #if DEBUG
158         std::cout << "LoadPFM: Could not open " << filename << std::endl;
159     }
160     std::string format;
161     int Width, Height;
162     float Endianess;
163     float ScaleFactor;
164     unsigned char r, g, b;
165     ifs >> format;
166     ifs >> Width >> Height;
167     ifs >> Endianess;
168     ScaleFactor = fabsf(Endianess);
169 // Little Endian
170     Bitmap result(Width, Height);
171     if (Endianess < 0)
172     {
173
174         std::string Line;
175         char ch;
176         unsigned char buffer[4];
177
178         while (ifs.peek() == '\n' || ifs.peek() == ' ')
179     {
180 #if DEBUG
181             std::cout << "PF: Ate some whitespace..." << std::endl;
182         #endif
183             ifs.read((char*)(&buffer[0]), 1);
184     }
185     for (int i = 0; i < Height; ++i)
186     {
187         for (int j = 0; j < Width; ++j)
188     {
189             float colour[3]{0,0,0};
190             for (int c = 0; c < 3; ++c)
191     {
192
193                 ifs.read((char*)(&buffer[0]), sizeof(buffer));
194 // DOS/IBM- ASCII HxD
195                 memcpy(&colour[c], &buffer, sizeof(float));
196             }
197 //std::cout << Colour(colour[0], colour[1], colour[2])
198 //            << std::endl;
199             result.SetPixel(j, i, Colour(colour[0], colour[1],
200                             colour[2]));
201         }
202     }
203 }
```

```

202     ifs.close();
203     return result;
204 #if DEBUG
205     std::cout << "Debug: Loaded " << filename << std::endl;
206 #endif
207 }
208
209 void Bitmap::SavePFM(std::string filename, float gamma) const
210 {
211     std::ofstream ofs(filename.c_str(), std::ios::out | std::ios::
212                         binary);
212     ofs << "PF" << std::endl;
213     ofs << Width << std::endl;
214     ofs << Height << std::endl;
215     ofs << "-1.0000" << std::endl;
216
217     Colour c;
218     float r, g, b;
219     unsigned char buffer[4];
220     for (int row = 0; row < Height; ++row)
221     {
222         for (int col = 0; col < Width; ++col)
223         {
224             c = Pixels[row][col];
225             r = c.GetR();
226             g = c.GetG();
227             b = c.GetB();
228             unsigned char const *x = reinterpret_cast<unsigned char
229                                     const *>(&r);
229             unsigned char const *y = reinterpret_cast<unsigned char
230                                     const *>(&g);
230             unsigned char const *z = reinterpret_cast<unsigned char
231                                     const *>(&b);
231             ofs << *x << *(x + 1) << *(x + 2) << *(x + 3);
232             ofs << *y << *(y + 1) << *(y + 2) << *(y + 3);
233             ofs << *z << *(z + 1) << *(z + 2) << *(z + 3);
234         }
235     }
236
237     ofs.close();
238 #if DEBUG
239     std::cout << "Debug PF: Saved " << filename << std::endl;
240 #endif
241 }
242
243 int Bitmap::GetWidth() const
244 {
245     return Width;
246 }
247 int Bitmap::GetHeight() const
248 {
249     return Height;

```


2.1.3 BRDF

Listing 2.4: BRDF header file

```
1 #pragma once
2
3 #include "../Core/Math/Vector.h"
4 #include "../Core/Math/OrthonormalBasis.h"
5 #include "Colour.h"
6 #include "Sampler.h"
7 #include "Sample.h"
8
9 #include <algorithm>
10 #include <map>
11 #include "Sampler/RandomSampler.h"
12 #include "Intersection.h"
13
14 class BRDF
15 {
16 public:
17     enum class Type;
18     friend BRDF::Type operator^(BRDF::Type a, BRDF::Type b);
19     friend BRDF::Type operator+(BRDF::Type a, BRDF::Type b);
20     friend BRDF::Type operator-(BRDF::Type a, BRDF::Type b);
21     friend BRDF::Type operator&(BRDF::Type a, BRDF::Type b);
22
23     BRDF(BRDF::Type type)
24         : BSDFType(type),
25           BSampler(new RandomSampler(1))
26     {}
27
28     // Defines the relationship between reflected radiance and incoming
     irradiance
29     virtual Colour F(const Vector& wo, const Vector& wi, Intersection&
     intersection) const = 0;
30
31     // Defines the relationship for reflectance for different
     directions.
32     // Used to return incoming direction wi from the viewing direction
     wo
33     virtual Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
     Intersection& intersection) const;
34
35     virtual float PDF(const Vector& wo, const Vector& wi, Intersection&
     intersection) const;
36
37     // Hemispherical-directional reflectance: Total reflection from
     light in a given direction over the hemisphere.
38     virtual Colour Rho(const Vector& wo, Intersection& intersection)
     const = 0;
39
40     static bool IsSameHemisphere(const Vector& u, const Vector& v)
41     {
```

```

42         return Dot(u, v) > 0.0f;
43     }
44
45     static bool IsBSDFType(BRDF::Type type, BRDF::Type flag)
46     {
47         return (flag & type) == flag;
48     }
49
50     Type GetType() const
51     { return BSDFType; }
52 protected:
53     BSampler* BSampler;
54     BRDF::Type BSDFType;
55 };
56
57 inline Colour BRDF::SampleF(const Vector& wo, Vector& wi, float& pdf,
58                             Intersection& intersection) const
59 {
60     //wi = Sample::CosineHemisphere(Point2D(rand() / float(RAND_MAX),
61                                         rand() / float(RAND_MAX)));
62     wi = Sample::CosineHemisphere(BSampler->Sample2D());
63     //wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, wi).Hat
64     //();
65     // flip so that on the same side of hemisphere
66     if (!IsSameHemisphere(Vector(intersection.GetNormal()), wi))
67     {
68         wi = -wi;
69     }
70     pdf = PDF(wo, wi, intersection);
71     return F(wo, wi, intersection);
72 }
73
74 inline float BRDF::PDF(const Vector& wo, const Vector& wi, Intersection
75 & intersection) const
76 {
77     if (IsSameHemisphere(Vector(intersection.GetNormal()), wi))
78     {
79         //Vector vi(OrthonormalBasis::ToLocal(intersection.ShadingBasis
80         //, wi).Hat());
81         return OrthonormalBasis::CosTheta(wi) / M_PI;
82     }
83     return 0.0f;
84     //std::cerr << "BRDF::PDF has not implemented." << std::endl;
85     //std::exit(-1);
86     //return 0.0f;
87 }
88
89 enum class BRDF::Type
90 {
91     NONE          = 0,
92     REFLECTION   = 1 << 0,
93     REFRACTION   = 1 << 1,

```

```

89     DIFFUSE      = 1 << 2,
90     SPECULAR    = 1 << 3,
91     GLOSSY      = 1 << 4,
92     EMISSIVE    = 1 << 5,
93     ALL          = REFLECTION + REFRACTION + DIFFUSE + SPECULAR + GLOSSY
94     ,
95 }
96 inline BRDF::Type operator+(BRDF::Type a, BRDF::Type b)
97 {
98     return static_cast<BRDF::Type>(static_cast<int>(a) | static_cast<
99         int>(b));
100 }
101 inline BRDF::Type operator-(BRDF::Type a, BRDF::Type b)
102 {
103     return static_cast<BRDF::Type>(static_cast<int>(a) & ~static_cast<
104         int>(b));
105 }
106 inline BRDF::Type operator&(BRDF::Type a, BRDF::Type b)
107 {
108     return static_cast<BRDF::Type>(static_cast<int>(a) & static_cast<
109         int>(b));
110 }
111 inline BRDF::Type operator^(BRDF::Type a, BRDF::Type b)
112 {
113     return static_cast<BRDF::Type>(static_cast<int>(a) ^ static_cast<
114         int>(b));

```

2.1.4 Camera

Listing 2.5: Camera header file

```
1 #pragma once
2
3 #include "../Core/Math/Vector.h"
4 #include "../Core/Math/Matrix.h"
5 #include "../Engine/Ray.h"
6
7 // A virtual camera defines a viewing area for rendering.
8 /*
9 * The Job of the camera is to generate a ray
10 */
11 class Camera
12 {
13 public:
14     Camera();
15     Camera(Point eye, Point target, Vector up);
16     Camera(Matrix& c2w);
17     //function CalculateRay which returns a ray towards the image plane
18     // x is the value along the U direction on the image
19     // y is the value along the V direction on the image
20     virtual Ray CalculateRay(float x, float y, float lensX, float lensY
21         , int width, int height) const = 0;
22     virtual void InitializeCamera();
23     virtual void UpdateCamera() {};
24
25 protected:
26     Vector U, V, W;      // Camera's local coordinate
27     Point Eye;           // Camera's origin
28     Point Target;        // Camera's view direction
29     Vector Up;
30     Matrix CameraToWorld;
31 };
```

Listing 2.6: Camera source file

```
1 #include "Camera.h"
2
3 Camera::Camera()
4     : Eye{ 0, 0, 0 }, Target{ 0, 0, -1 }, Up{ 0, 1, 0 }
5 {}
6
7 Camera::Camera(Point eye, Point target, Vector up)
8     : Eye{ eye }, Target{ target }, Up{ up }
9 {}
10
11 Camera::Camera(Matrix& c2w)
12     : CameraToWorld(c2w)
13 {}
14
15 void Camera::InitializeCamera()
16 {
17     // Calculate orthonormal basis U, V, W (Camera-World Transformation
18     // )
19     W = (Eye - Target).Hat();
20     U = Cross(Up, W).Hat();
21     V = Cross(W, U);
22     //CameraToWorld = Matrix(U, V, W);
23     //CameraToWorld[11] = -Eye.GetX();
24     //CameraToWorld[12] = -Eye.GetY();
25     //CameraToWorld[13] = -Eye.GetZ();
25 }
```

2.1.5 Colour

Listing 2.7: Colour header file

```
1 #pragma once
2
3 #include <cmath>
4 #include <iostream>
5 #include "../Core/Math/Vector.h"
6 #include "../Core/Utility/Misc.h"
7
8 class Colour
9 {
10 public:
11     Colour();
12     Colour(float c);
13     Colour(float r, float g, float b);
14     Colour(const Colour& colour);
15     Colour(const Vector& colour);
16
17     float Average() const;
18
19     Colour Clamp() const;
20     Colour operator+(const Colour& c) const;
21     Colour operator-(const Colour& c) const;
22     Colour operator*(const Colour& c) const;
23     Colour operator/(const Colour& c) const;
24
25     Colour operator*(float s) const;
26     Colour operator/(float s) const;
27
28     Colour& operator+=(const Colour& c);
29     Colour& operator-=(const Colour& c);
30     Colour& operator*=(const Colour& c);
31     Colour& operator*=(float s);
32     Colour& operator/=(float s);
33
34     // Exponentiation operator
35     Colour operator^(float s);
36     Colour& operator^=(float s);
37     Colour operator-() const;
38
39     bool operator==(const Colour& c) const;
40     bool operator!=(const Colour& c) const;
41     bool operator<(const Colour& c) const;
42
43     Colour& operator=(const Colour& c);
44
45     float MaxComponent() const;
46
47     float& Colour::operator[](int index);
48     float Colour::operator[](int index) const;
```

```

50     friend Colour operator*(float s, const Colour& c);
51     friend std::ostream& operator<<(std::ostream& os, const Colour& c);
52     friend std::istream& operator>>(std::istream& is, Colour& c);
53
54     unsigned char rgb() const;
55
56     float GetR() const;
57     float GetG() const;
58     float GetB() const;
59
60     void SetR(float r);
61     void SetG(float g);
62     void SetB(float b);
63     static Colour Exp(Colour c);
64 private:
65     float R, G, B;
66 };
67
68
69 inline float Colour::GetR() const
70 {
71     return R;
72 }
73
74 inline float Colour::GetG() const
75 {
76     return G;
77 }
78
79 inline float Colour::GetB() const
80 {
81     return B;
82 }
83
84 inline void Colour::SetR(float nx)
85 {
86     R = nx;
87 }
88
89 inline void Colour::SetG(float ny)
90 {
91     G = ny;
92 }
93
94 inline void Colour::SetB(float nz)
95 {
96     B = nz;
97 }

```

Listing 2.8: Colour source file

```
1 #include "Colour.h"
2
3 Colour::Colour()
4     : R{ 0 }, G{ 0 }, B{ 0 }
5 {}
6
7 Colour::Colour(float c)
8     : R(c), G(c), B(c)
9 {}
10
11 Colour::Colour(float R, float G, float B)
12     : R{ R }, G{ G }, B{ B }
13 {}
14
15 Colour::Colour(const Colour& colour)
16     : R( colour.R ), G( colour.G ), B(colour.B)
17 {}
18
19 Colour::Colour(const Vector& v)
20     : R{ v.GetX() }, G{ v.GetY() }, B{ v.GetZ() }
21 {}
22
23 float Colour::Average() const
24 {
25     return (R + G + B) / 3;
26 }
27
28 Colour Colour::Clamp() const
29 {
30     return Colour(clamp(R, 0.0f, 1.0f), clamp(G, 0.f, 1.f), clamp(B, 0.
31         f, 1.f));
32 }
33
34 Colour Colour::operator+(const Colour& c) const
35 {
36     return Colour(R + c.R, G + c.G, B + c.B);
37 }
38
39 Colour Colour::operator-(const Colour& c) const
40 {
41     return Colour(R - c.R, G - c.G, B - c.B);
42 }
43
44 Colour& Colour::operator+=(const Colour& c)
45 {
46     R += c.R;
47     G += c.G;
48     B += c.B;
49     return *this;
50 }
```

```

51 Colour& Colour::operator==(const Colour& c)
52 {
53     R -= c.R;
54     G -= c.G;
55     B -= c.B;
56     return *this;
57 }
58
59 Colour Colour::operator*(const Colour& c) const
60 {
61     return Colour(R*c.R, G*c.G, B*c.B);
62 }
63
64 Colour Colour::operator/(const Colour& c) const
65 {
66     return Colour(R / c.R, G / c.G, B / c.B);
67 }
68
69 Colour& Colour::operator*=(const Colour& c)
70 {
71     R *= c.R;
72     G *= c.G;
73     B *= c.B;
74     return *this;
75 }
76
77
78 Colour Colour::operator*(float s) const
79 {
80     return Colour(R*s, G*s, B*s);
81 }
82
83 Colour& Colour::operator*=(float s)
84 {
85     R *= s;
86     G *= s;
87     B *= s;
88     return *this;
89 }
90
91 Colour Colour::operator/(float s) const
92 {
93     s = 1 / s;
94     return Colour(R * s, G * s, B * s);
95 }
96
97 Colour& Colour::operator/=(float s)
98 {
99     s = 1 / s;
100    R *= s;
101    G *= s;
102    B *= s;

```

```

103     return *this;
104 }
105
106 float& Colour::operator[](int index)
107 {
108     return *(&R + index);
109 }
110
111 float Colour::operator[](int index) const
112 {
113     return *(&R + index);
114 }
115
116 Colour Colour::operator^(float s)
117 {
118     return Colour(powf(R, s), powf(G, s), powf(B, s));
119 }
120
121 Colour& Colour::operator^=(float s)
122 {
123     R = powf(R, s);
124     G = powf(G, s);
125     B = powf(B, s);
126     return *this;
127 }
128
129 Colour Colour::operator-() const
130 {
131     return Colour(-R, -G, -B);
132 }
133
134 bool Colour::operator==(const Colour& c) const
135 {
136     return (R == c[0] && G == c[1] && B == c[2]);
137 }
138
139 bool Colour::operator!=(const Colour& c) const
140 {
141     return !(*this == c);
142 }
143
144 bool Colour::operator<(const Colour& c) const
145 {
146     return (this->R < c.R && this->G < c.G && this->B < c.B);
147 }
148
149 Colour& Colour::operator=(const Colour& c)
150 {
151     if (this == &c)
152         return *this;
153     R = c.R;
154     G = c.G;

```

```

155     B = c.B;
156
157     return *this;
158 }
159
160 float Colour::MaxComponent() const
161 {
162     return Max(R, Max(G, B));
163 }
164
165 Colour operator*(float s, const Colour& c)
166 {
167     return Colour(s * c.R, s * c.G, s * c.B);
168 }
169
170 std::ostream& operator<<(std::ostream& os, const Colour& v)
171 {
172     os << "Colour(" << v.R << ", " << v.G << ", " << v.B << ")";
173     return os;
174 }
175
176 std::istream& operator>>(std::istream& is, Colour& v)
177 {
178     is >> v.R >> v.G >> v.B;
179     return is;
180 }
181
182
183 unsigned char Colour::rgb() const
184 {
185     unsigned char r = R * 255;
186     unsigned char g = G * 255;
187     unsigned char b = B * 255;
188     return r + g + b;
189 }
190
191 Colour Colour::Exp(Colour c)
192 {
193     return Colour(std::exp(c[0]), std::exp(c[1]), std::exp(c[2]));
194 }

```

2.1.6 Geometry

Listing 2.9: Geometry header file

```
1 #pragma once
2 #include "Ray.h"
3 #include "Intersection.h"
4 #include "BBox.h"
5 #include "../Core/Math/Transform.h"
6 class Point2D;
7
8 // Geometry represents a object that can intersected with a ray.
9 class Geometry
10 {
11 public:
12
13     Geometry()
14     {}
15
16     Geometry(Transform* ObjToWorld, Transform* WorldToObject)
17         : ObjectToWorld(ObjToWorld), WorldToObject(WorldToObject)
18     {}
19     virtual ~Geometry() {}
20
21     virtual float SurfaceArea() const = 0; // Used for area lights
22     virtual bool IntersectRay(const Ray& ray, float& tMin, float&
23         epsilon, Intersection& intersection) const = 0;
24     virtual BBox GetBounds() const { return BoundingBox; }
25     virtual bool IsCompound() const { return false; }
26     virtual int GetNumberOfGeometries() { return 1; }
27     virtual Geometry* GetSubGeometry(int i) { return nullptr; }
28     virtual Point Sample(const Point2D& sample, Normal& normal) const {
29         std::cerr << "Shape function <Sample> not implemented";
30         return Point(0, 0, 0); }
31     // Pdf defined over area
32     virtual float Pdf(const Point& sample) const { return 1.0f /
33         SurfaceArea(); }
34     Transform* GetObjectToWorld() const
35     { return ObjectToWorld; }
36     // Intersection for shadow rays
37     //virtual bool IntersectRay(const Ray& ray) const = 0;
38 protected:
39     BBox BoundingBox;
40     Transform* ObjectToWorld;
41     Transform* WorldToObject;
42 }
```

2.1.7 Integrator

Listing 2.10: Integrator header file

```
1 #pragma once
2 #include "Sampler.h"
3 class Scene;
4
5 class Integrator
6 {
7 public:
8     virtual void Preprocess(Scene* scene, Sampler* sampler) { };
9     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
10                     sampler) const = 0;
11 }
```

2.1.8 Intersection

Listing 2.11: Intersection header file

```
1 #pragma once
2
3 #include "Ray.h"
4 #include "../Core/Math/Point.h"
5 #include "../Core/Math/Normal.h"
6 #include "../Core/Math/OrthonormalBasis.h"
7 class Geometry;
8 class Primitive;
9 class Transform;
10 class Material;
11
12 // Class contains information about the intersection
13 struct Intersection
14 {
15 public:
16     const Point GetPoint() { return IPoint; }
17     inline Normal GetNormal() { return INormal; }
18     void SetPoint(Point& point)
19     {IPoint = point; }
20     void SetNormal(const Normal& normal)
21     {INormal = normal; }
22
23     Material* material;
24     Point    IPPoint; // hit point
25     Normal   INormal; // normal at the point of intersection
26     Normal N;
27     float U, V; // Interpolated UV
28     float U0, V0; // UV
29     Geometry* IGeometry; // Pointer to the object it hit.
30
31     OrthonormalBasis ShadingBasis;
32
33     Transform* worldToObject;
34     Transform* objectToWorld;
35     Primitive* primitive;
36 };
```

2.1.9 Mapping

Listing 2.12: Mapping header file

```
1 #pragma once
2
3 #include "../Core/Utility/Misc.h"
4 #include "../Core/Math/Vector.h"
5 #include "../Core/Math/Point2D.h"
6
7 class Mapping
8 {
9 public:
10     static Point2D Spherical(Vector& p)
11     {
12         float phi = SphericalPhi(p);
13         float theta = SphericalTheta(p);
14         float u = phi / (2 * M_PI);
15         float v = theta * M_INVPI;
16         assert(v >= 0.0);
17         assert(u >= 0.0);
18         return Point2D(u, v);
19     }
20
21     static Point2D Angular(Vector& v)
22     {
23         //return Point2D(abs(cos(SphericalPhi(v))), fabsf(sin(
24             //SphericalTheta(v))));
25         //return Point2D(0, 0);
26         Vector vv = Normalize(v);
27         //float d = sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY());
28         //float r = d ? 0.159154943 * acos(v.GetZ()) / d : 0;
29         float r = 0.159154943*acos(vv.GetZ()) / sqrt(vv.GetX()*vv.GetX()
30             () + vv.GetY()*vv.GetY());
31         return Point2D(0.5 + v.GetX() * r, 0.5 + v.GetY() * r);
32     }
33 };
```

2.1.10 Material

Listing 2.13: Material header file

```
1 #pragma once
2
3 #include "../Core/Math/Vector.h"
4 #include "BRDF.h"
5 #include "Texture.h"
6 #include "Intersection.h"
7 #include "Primitive.h"
8
9 class Material
10 {
11 public:
12     virtual ~Material()
13     {
14     }
15
16     Material()
17     {}
18
19     virtual Colour F(Intersection& Point, Vector Wo, Vector Wi) const =
20         0;
21     virtual Colour SampleF(Intersection& intersection, Vector Wo,
22                             Vector& Wi, float& Pdf) const
23     {
24         Pdf = 0.0f;
25         return 0.0f;
26     }
27
28
29     virtual BRDF::Type GetType() const = 0;
30     virtual bool IsTranslucent() const;
31 protected:
32
33     static void NormalBump(Intersection& intersection, Texture*
34                             bumpTexture, bool flipRed = false, bool flipGreen = false,
35                             float bumpAmount = 1.0f)
36     {
37         auto pixel(bumpTexture->GetValue(intersection.U, intersection.V
38                                             , intersection));
39         float redChannel = pixel.GetR();
40         float greenChannel = pixel.GetG();
41         if (flipRed)
42         {
43             redChannel = 1 - redChannel;
44         }
45         if (flipGreen)
46         {
```

```
44         greenChannel = 1 - greenChannel;
45     }
46
47     auto normalVector = bumpAmount * Vector(2.0f * redChannel - 1,
48 #if 0
49         2.0f * greenChannel - 1, 2.0f * pixel.GetB() - 1);
50     if (Dot(Normal(OrthonormalBasis::ToLocal(intersection.
51         ShadingBasis, normalVector)), intersection.INormal) < 0.0f)
52     {
53     normalVector = -normalVector;
54     }
55     intersection.INormal = (intersection.INormal + Normal(
56         OrthonormalBasis::ToLocal(intersection.ShadingBasis,
57         normalVector))).Hat();
58     intersection.ShadingBasis = OrthonormalBasis::FromW(Vector(
59         intersection.INormal));
60 }
61 inline bool Material::IsTranslucent() const
62 {
63     return false;
64 }
```

2.1.11 Photon

Listing 2.14: Photon header file

```
1 #pragma once
2
3 #include "../Core/Math/Vector.h"
4 #include "Colour.h"
5 #include <vector>
6 #include <algorithm>
7 #include "BBox.h"
8 #include "../Core/Utility/Timer.h"
9
10 struct Photon
11 {
12     Point position;
13     Vector incidentDirection;
14     Colour power;
15
16     friend std::ostream& operator<<(std::ostream& os, const Photon& p);
17     friend std::istream& operator>>(std::istream& is, Photon& photon);
18
19 };
20
21 inline std::ostream& operator<<(std::ostream& os, const Photon& photon)
22 {
23     os << std::fixed << photon.position[0] << " " << photon.position[1]
24         << " " << photon.position[2] << " ";
25     os << std::fixed << photon.incidentDirection[0] << " " << photon.
26         incidentDirection[1] << " " << photon.incidentDirection[2] << "
27         ";
28     os << std::fixed << photon.power[0] << " " << photon.power[1] << "
29         " << photon.power[2] << std::endl;
30     return os;
31 }
32
33 inline std::istream& operator>>(std::istream& is, Photon& photon)
34 {
35     is >> photon.position;
36     is >> photon.incidentDirection;
37     is >> photon.power;
38     return is;
39 }
40
41 struct KDTreeNode
42 {
43     Photon photon;
44     int axis;
45     KDTreeNode* left = nullptr;
46     KDTreeNode* right = nullptr;
47     bool IsLeaf() const
```

```

46     {
47         return (left == nullptr && right == nullptr);
48     }
49 };
50
51
52 struct KDTreeCompare
53 {
54     int axis;
55
56     KDTreeCompare(int axis)
57         :axis(axis){}
58
59     bool operator()(Photon& a, Photon& b) const
60     {
61         return a.position[axis] < b.position[axis];
62     }
63 };
64
65
66 struct KDTreeResult
67 {
68     Photon photon;
69     float distanceSquared;
70
71     KDTreeResult(Photon p, float distance)
72     {
73         photon = p;
74         distanceSquared = distance;
75     }
76
77
78 };
79
80
81 struct KDTreeQuery
82 {
83     float searchDistanceSquared;
84     Point queryPosition;
85     int maxNumber;
86     int found = 0;
87     std::vector<KDTreeResult> result;
88 };
89
90
91 struct KDTreeResultComparator : std::binary_function<KDTreeResult&, KDTreeResult&, boolbool operator()(KDTreeResult& a, KDTreeResult& b) const
94     {
95         return a.distanceSquared < b.distanceSquared;
96     }

```

```

97  };
98
99
100 class KDTree
101 {
102 public:
103     KDTree()
104         :root(new KDTreeNode)
105     {
106     }
107 }
108
109     KDTree(std::vector<Photon> pho)
110         : root(new KDTreeNode),
111           photons(photons)
112     {
113
114         build(root, pho);
115     }
116
117     void build() const
118     {
119         if (photons.size() == 0)
120         {
121             return;
122         }
123         build(root, photons);
124     }
125
126     void build(KDTreeNode* node, std::vector<Photon> photons) const
127     {
128
129         BBox volume;
130         for (int i = 0; i < photons.size(); ++i)
131         {
132             volume = volume + photons[i].position;
133         }
134
135         int splitAxis = volume.LongestAxis();
136
137         if (photons.size() == 1)
138         {
139             node->axis = splitAxis;
140             node->photon = photons[0];
141             return;
142         }
143
144         int splitPos = floor(photons.size() / 2.0);
145         std::nth_element(photons.begin(), photons.begin() + splitPos,
146                         photons.end(), KDTreeCompare(splitAxis));
147         node->photon = photons[splitPos];
148         node->axis = splitAxis;

```

```

148
149     std::vector<Photon> left(photons.begin(), photons.begin() +
150         splitPos);
150     node->left = new KDTreeNode();
151     build(node->left, left);
152     if (photons.begin() + splitPos + 1 < photons.end())
153     {
154         std::vector<Photon> right(photons.begin() + splitPos + 1,
155             photons.end());
155         node->right = new KDTreeNode();
156         build(node->right, right);
157     }
158 }
160
161 void find(KDTreeQuery& query) const
162 {
163     findHelper(root, query);
164 }
165
166
167 void findNN(KDTreeQuery& query) const
168 {
169     findNNHelper(root, query);
170 }
171
172
173 friend std::ostream& operator<<(std::ostream& os, const KDTree&
174     tree)
175 {
176     tree.print(os, tree.root);
177     return os;
178 }
179
180 void scalePhotonPw(float s, int a, int b)
181 {
182     for (int i = a; i < b; ++i)
183     {
184         photons[i].power *= s;
185     }
186 }
187
188 void clear()
189 {
190     if (root)
191     {
192         if (root->left)
193         {
194             delete root->left;
195         }
196         if (root->right)
197         {
198

```

```

197             delete root->right;
198         }
199         delete root;
200     }
201     root = new KDTreeNode();
202 }
203
204 friend std::istream& operator>>(std::istream& is, KDTree& tree)
205 {
206     tree.clear();
207     Photon photon;
208     while (is >> photon)
209     {
210         tree.push_back(photon);
211     }
212     tree.build();
213     return is;
214 }
215
216 void print(std::ostream& os, KDTreeNode* node) const
217 {
218     if (!node)
219     {
220         return;
221     }
222     os << node->photon.position << std::endl;
223     if (node->left)
224     {
225         os << "left ->" << std::endl;
226         print(os, node->left);
227     }
228     if (node->right)
229     {
230         os << "right ->" << std::endl;
231         print(os, node->right);
232     }
233 }
234
235 void push_back(Photon& p)
236 {
237     photons.push_back(p);
238 }
239
240
241 size_t size() const
242 {
243     return photons.size();
244 }
245
246
247 private:
248     void findHelper(KDTreeNode* node, KDTreeQuery& query) const

```



```

296         findNNHelper(node->right, query);
297         // hypersphere lies on both sides
298         if (shouldSearchBoth)
299             findNNHelper(node->left, query);
300     }
301     else
302     {
303         findNNHelper(node->left, query);
304         if (shouldSearchBoth)
305             findNNHelper(node->right, query);
306     }
307
308     float dist2 = DistanceSquared(query.queryPosition, node->photon
309         .position);
310     if (dist2 < query.searchDistanceSquared)
311     {
312         if (query.result.size() < query.maxNumber)
313         {
314             query.result.push_back(KDTreeResult(node->photon, dist2
315                 ));
316             query.found++;
317         }
318         else
319         {
320             std::sort(query.result.begin(), query.result.end(),
321                 KDTreeResultComparator());
322             //for (int i = 0; i < query.result.size(); ++i)
323             //{
324             //    std::cout << query.result[i].photon.position << std
325                 ::endl;
326             //}
327             //std::cout << std::endl;
328             // check if point distance is less than the furthest
329             // point
330
331             if (dist2 < query.result.back().distanceSquared)
332             {
333                 query.result.pop_back();
334                 query.result.push_back(KDTreeResult(node->photon,
335                     dist2));
336             }
337         }
338     }
339     public:
340     KDTreeNode* root;
341     std::vector<Photon> photons;
342 };

```

2.1.12 Primitive

Listing 2.15: Primitive header file

```
1 #pragma once
2 #include "Material.h"
3 #include "BBox.h"
4
5 class Primitive
6 {
7 public:
8     virtual ~Primitive()
9     {
10    }
11
12     virtual bool IntersectRay(const Ray& ray, float& tMin, float&
13         epsilon, Intersection& intersection) const = 0;
14     virtual bool IsEmissive() const { return false; }
15     virtual bool IsCompound() const { return false; }
16     virtual BBox GetBounds() const = 0;
17     virtual Material* GetMaterial() const = 0;
18     virtual std::vector<Primitive*> GetPrimitive() const;
19     virtual Point Sample(const Point2D& sample, Normal& normal) const {
20         return Point(0,0,0); }
21
22     inline std::vector<Primitive*> Primitive::GetPrimitive() const
23     {
24         throw std::runtime_error("Calling GetPrimitive from Primitive");
25         return {};
26     }
```

2.1.13 Ray

Listing 2.16: Ray header file

```
1 #pragma once
2
3 #include "../Core/Math/Point.h"
4 #include "../Core/Math/Vector.h"
5 #include <cmath>
6
7 // Note that rays are always defined in world space.
8 class Ray
9 {
10 public:
11     Ray();
12     Ray(const Point& Origin, const Vector& Direction, float MinT = 1e
13          -6, float MaxT = INFINITY, int Depth = 0);
13     Ray(const Ray& other);
14
15     inline const Vector& GetDirection() const;
16     inline const Point& GetOrigin() const;
17     inline int GetDepth() const;
18     Point operator()(float t) const;
19     friend std::ostream& operator<<(std::ostream& os, const Ray& ray);
20     bool IsRefractiveRay() const;
21     mutable float MinT;
22     mutable float MaxT;
23     mutable int Depth;
24     mutable bool Refractive;
25 private:
26     Point Origin;
27     Vector Direction;
28 };
29
30 inline const Vector& Ray::GetDirection() const
31 {
32     return Direction;
33 }
34
35 inline const Point& Ray::GetOrigin() const
36 {
37     return Origin;
38 }
39
40 inline int Ray::GetDepth() const
41 {
42     return Depth;
43 }
```

Listing 2.17: Ray source file

```
1 #include "Ray.h"
2
3 Ray::Ray()
4     :Origin{}, Direction{}, Depth(0), MinT(FLT_EPSILON), MaxT(INFINITY)
5         , Refractive(false)
6 {}
7
8 Ray::Ray(const Point& Origin, const Vector& Direction, float MinT,
9     float MaxT, int Depth)
10    : Origin{Origin}, Direction{Direction}, MinT(MinT), MaxT(MaxT),
11        Depth(Depth), Refractive(false)
12 {}
13
14 Ray::Ray(const Ray& other)
15     : Origin(other.Origin),
16         Direction(other.Direction),
17         MinT(other.MinT),
18         MaxT(other.MaxT),
19         Depth(other.Depth),
20         Refractive(other.Refractive)
21 {
22 }
23
24 Point Ray::operator()(float t) const
25 {
26     return Origin + t * Direction;
27 }
28
29 bool Ray::IsRefractiveRay() const
30 {
31     return Refractive;
32 }
33
34 std::ostream& operator<<(std::ostream& os, const Ray& ray)
35 {
36     os << "Ray<" << ray.Origin << "+" << ray.Direction << ">";
37     return os;
38 }
```

2.1.14 Renderer

Listing 2.18: Renderer header file

```
1 #pragma once
2
3 #include <functional>
4 #include "Scene.h"
5 #include "Bitmap.h"
6 class Renderer
7 {
8 public:
9     typedef std::function<void(int x, int y, int w, int h, Bitmap* b)>
10        UpdateScreenCallback;
11     typedef std::function<void(int x, int y, int w, int h)>
12        StartScreenCallback;
13     virtual void Render() const = 0;
14     //virtual Colour Li(const Ray& ray, Sampler* sampler, Intersection&
15     //intersect) const = 0;
16
17     virtual void OnUpdate(int x, int y) const
18     {}
19
20     virtual void OnComplete() const {}
21     virtual void OnStop() {}
22     virtual Bitmap* GetBitmap() { return nullptr; }
23
24 protected:
25     UpdateScreenCallback func = nullptr;
26     StartScreenCallback start_func = nullptr;
27 };
```

2.1.15 Sample

Listing 2.19: Sample header file

```
1 #pragma once
2
3 #include "../Core/Math/Point2D.h"
4 #include "../Core/Math/Vector.h"
5 #include "../Core/Utility/Misc.h"
6 #include "../Core/Utility/RNG.h"
7 #include "../Engine/Sampler.h"
8 #include <algorithm>
9
10 namespace Sample
11 {
12     static void DiscRejectionSample(Random::RandomDouble& rng, Point2D&
13                                     f)
14     {
15         float x = rng(), y = rng();
16         while (x * x + y * y > 1.0)
17         {
18             x = rng();
19             y = rng();
20         }
21         f.SetX(x);
22         f.SetY(y);
23     }
24     static Vector UniformHemisphereSample(Random::RandomDouble& rng,
25                                           const Normal& normal)
26     {
27         Vector v;
28         do
29         {
30             v.SetX(1.f - 2.f * rng());
31             v.SetY(1.f - 2.f * rng());
32             v.SetZ(1.f - 2.f * rng());
33         } while (v.LengthSquared() > 1.0f);
34
35         if (Dot(v, normal) < 0.0f)
36         {
37             v = -v;
38         }
39         v.Hat();
40         return v;
41     }
42     static void DiscPolarSample(Point2D& f, float radius)
43     {
44         float Phi = 2 * M_PI * f.GetX();
45         float r = radius * sqrt(f.GetY());
46         f.SetX(r * cos(Phi)*0.5 + 0.5);
47         f.SetY(r * sin(Phi)*0.5 + 0.5);
```

```

48     }
49
50 // https://mediatech.aalto.fi/~jaakko/T111-5310/K2013/JGT-97.pdf
51 static Point2D DiscConcentricSample(const Point2D& f)
52 {
53     float phi, r, u, v;
54     float a = 2 * f.GetX() - 1;
55     float b = 2 * f.GetY() - 1;
56     if (a > -b)
57     {
58         if (a > b)
59         {
60             r = a;
61             phi = (M_PI / 4) * (b/a);
62         }
63         else
64         {
65             r = b;
66             phi = (M_PI / 4) * (2 - (a/b));
67         }
68     }
69     else
70     {
71         if (a < b)
72         {
73             r = -a;
74             phi = (M_PI / 4) * (4 + (b/a));
75         }
76         else
77         {
78             r = -b;
79             if (b != 0)
80                 phi = (M_PI / 4) * (6 - (a/b));
81             else
82                 phi = 0;
83         }
84     }
85     u = r * cos(phi);
86     v = r * sin(phi);
87     return Point2D(u, v);
88 }
89
90 static Vector HemisphereRejectionSample(Random::RandomDouble& rng)
91 {
92     float x = rng(), y = rng(), z = rng();
93     while (x * x + y * y + z * z > 1.0)
94     {
95         x = rng();
96         y = rng();
97         z = rng();
98     }
99     return Vector(x,y,z);

```

```

100    }
101
102    static Vector HemisphereSample(const Point2D& f)
103    {
104        // Use spherical coordinates
105        double Radius = sqrt(1.0 - f.GetX() * f.GetX());
106        double Phi = 2 * M_PI * f.GetY();
107        return Vector(cos(Phi) * Radius, sin(Phi)* Radius, f.GetX());
108    }
109
110    static Point2D CosineHemisphereSample(Point2D& f)
111    {
112        // theta, phi, spherical coordinates
113        return Point2Dacos(sqrt(f.GetX())), 2 * M_PI * f.GetY());
114    }
115
116    static Vector UniformSphericalCap(const Point2D& p, float
117        cosThetaMax)
118    {
119        float z = (1 - p[0] * cosThetaMax) * 2 - 1;
120        float r = sqrtf(std::max(0.0f, 1.0f - z * z));
121        float phi = 2.0f * M_PI * p[1];
122        float x = r * cosf(phi);
123        float y = r * sinf(phi);
124        return Vector(x, z, y);
125    }
126
127    static float UniformSphericalCapPDF(float cosThetaMax)
128    {
129        return 1.0f / (4.0f * M_PI * cosThetaMax);
130    }
131
132    static Vector UniformSphere(const Point2D& p)
133    {
134        float z = p[0] * 2 - 1;
135        float r = sqrtf(std::max(0.0f, 1.0f - z * z));
136        float phi = 2.0f * M_PI * p[1];
137        float x = r * cosf(phi);
138        float y = r * sinf(phi);
139        return Vector(x, y, z);
140    }
141
142    static float UniformSpherePDF()
143    {
144        return 1.0f / (4.0f * M_PI);
145    }
146
147    static Vector UniformHemisphere(const Point2D& p)
148    {
149        float z = p[0];
150        float r = sqrtf(std::max(0.0f, 1.0f - z * z));

```

```

151     float x = r * cosf(phi);
152     float y = r * sinf(phi);
153     return Vector(x, y, z);
154 }
155
156 static float UniformHemispherePDF()
157 {
158     return 1.0f / (2.0f * M_PI);
159 }
160
161 static Vector CosineHemisphere(const Point2D& p)
162 {
163     float z = sqrtf(p[0]);
164     float r = sqrtf(std::max(0.0f, 1.0f - z * z));
165     float phi = 2.0f * M_PI * p[1];
166     float x = r * cosf(phi);
167     float y = r * sinf(phi);
168     return Vector(x, y, z);
169 }
170
171 static float CosineHemispherePDF(float cosTheta)
172 {
173     return cosTheta / M_PI ;
174 }
175
176 static Point2D UniformTriangle(const Point2D& p)
177 {
178     float s = sqrtf(p[0]);
179     float x = 1.0f - s;
180     float y = p[1] * s;
181     return Point2D(x, y);
182 }
183
184 // https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf
185 static Vector Beckmann(const Point2D &sample, float alpha) {
186     float theta = atan(sqrt((-alpha*alpha) * log(1 - sample[0])));
187     float z = cos(theta);
188     float r = sqrtf(std::max(0.0f, 1.0f - z * z));
189     float phi = 2.0f * M_PI * sample[1];
190     float x = r * cosf(phi);
191     float y = r * sinf(phi);
192
193     return Vector(x, y, z);
194 }
195
196 static float BeckmannPdf(const Vector &m, float alpha) {
197     float costheta = OrthonormalBasis::CosTheta(m);
198     if (costheta > 0.0f)
199     {
200         float theta = acos(m[2]);
201         float t = -powf(tan(theta), 2.0f) / (alpha * alpha);
202         float e = exp(t);

```

```

203         float d = M_PI * alpha * alpha * powf(costheta, 4.0f);
204         return (e / d) * costheta;
205     }
206     return 0.0f;
207 }
208
209 static Vector Phong(const Point2D &sample, float alpha)
210 {
211     //alpha = 2 * sqrtf(alpha) - 2;
212     float theta = acosf(powf(sample[0], 1 / (alpha + 2)));
213     float z = cos(theta);
214     float r = sqrtf(std::max(0.0f, 1.0f - z * z));
215     float phi = 2.0f * M_PI * sample[1];
216     float x = r * cosf(phi);
217     float y = r * sinf(phi);
218
219     return Vector(x, y, z);
220 }
221
222 static float PhongPdf(const Vector &m, float alpha)
223 {
224     float costheta = OrthonormalBasis::CosTheta(m);
225     if (costheta > 0.0f)
226     {
227         //alpha = 2 * sqrtf(alpha) - 2;
228         float a = (alpha + 2) / (2 * M_PI);
229         return a * costheta * powf(costheta, alpha);
230     }
231     return 0.0f;
232 }
233
234 static Vector GGX(const Point2D& sample, float alpha)
235 {
236     float theta = atanf(alpha * sqrtf(sample[0]) / sqrtf(1.0f -
237         sample[0]));
238     float phi = 2 * M_PI * sample[1];
239     float z = cos(theta);
240     float sintheta = sqrtf(std::max(0.0f, 1.0f - z * z));
241     float x = sintheta * cosf(phi);
242     float y = sintheta * sinf(phi);
243     return Vector(x, y, z);
244 }
245
246 static float GGXPdf(const Vector& m, float alpha)
247 {
248     if (m[2] > 0.0f)
249     {
250         float costheta = m[2];
251
252         float alpha2 = alpha * alpha;
253         float e = alpha2 + powf(tanf(acosf(costheta)), 2.0f);
254         float d = M_PI * powf(costheta, 3.0f) * e * e;

```

```
254         return alpha2 / d;  
255     }  
256     return 0.0f;  
257 }  
258 };
```

2.1.16 Sampler

Listing 2.20: Sampler header file

```
1 #pragma once
2
3 #include <random>
4 #include "../Core/Math/Point2D.h"
5 #include "../Core/Math/Vector.h"
6 #include <vector>
7 #include <string>
8 #include <fstream>
9 #include <iostream>
10 class Sampler
11 {
12 public:
13     // samples - Number of sub pixel samples
14     Sampler(int samples)
15         : NumberOfSamples(samples){}
16
17     // Generates samples in a unit square
18     virtual void GenerateSamples() = 0;
19
20     // Get next 2D sample in unit square
21     virtual Point2D Sample2D() const = 0;
22
23     // Get next 1D sample in unit square
24     virtual float Sample1D() const = 0;
25
26     inline int GetNumberOfSamples();
27
28     static void Output(std::vector<Vector>& Samples, std::string path)
29     {
30         std::ofstream ofs(path);
31         std::stringstream XCoordinates;
32         std::stringstream YCoordinates;
33         std::stringstream ZCoordinates;
34
35         ofs << "clf;" << std::endl;
36         ofs << "graphics_toolkit(\"gnuplot\")" << std::endl;
37
38         XCoordinates << "x = [";
39         YCoordinates << "y = [";
40         ZCoordinates << "z = [";
41
42         if (Samples.size() > 1)
43         {
44             Vector s = Samples[0];
45             XCoordinates << s.GetX();
46             YCoordinates << s.GetY();
47             ZCoordinates << s.GetZ();
48         }
49     }
```

```

50     for (int i = 1; i < Samples.size(); ++i)
51     {
52         Vector s = Samples[i];
53         XCoordinates << ", " << s.GetX();
54         YCoordinates << ", " << s.GetY();
55         ZCoordinates << ", " << s.GetZ();
56     }
57     XCoordinates << "];" << std::endl;
58     YCoordinates << "];" << std::endl;
59     ZCoordinates << "];" << std::endl;
60
61     ofs << XCoordinates.str() << std::endl;
62     ofs << YCoordinates.str() << std::endl;
63     ofs << ZCoordinates.str() << std::endl;
64
65     ofs << "c = cos(x .* y) .* z;" << std::endl;
66     ofs << "p1 = scatter3 (x, y, z, 8, c, 'filled');" << std::endl;
67     ofs << "title ('Hemisphere mapping with " << Samples.size() <<
68         " samples');" << std::endl;
69     ofs << "hold on" << std::endl;
70     ofs << "theta = 2 * pi;" << std::endl;
71     ofs << "phi = pi / 2;" << std::endl;
72     ofs << "u = linspace(0, theta, 16); v = linspace(0, phi, 16);"
73         << std::endl;
74     ofs << "[Theta, Phi] = meshgrid(u, v);" << std::endl;
75     ofs << "p2 = surf(cos(Theta).*sin(Phi), sin(Phi).*sin(Theta),"
76         "cos(Phi), 'FaceAlpha', 0);" << std::endl;
77     ofs << "grid on;" << std::endl;
78     // Sets major axis ticks to have correct spacing
79     float spacing = sqrtf(Samples.size()) / Samples.size();
80     ofs << "xbounds = xlim();" << std::endl;
81     ofs << "set(gca, 'xtick', xbounds(1):" << spacing << ":xbounds"
82         "(2));" << std::endl;
83     ofs << "ybounds = ylim();" << std::endl;
84     ofs << "set(gca, 'ytick', ybounds(1):" << spacing << ":ybounds"
85         "(2));" << std::endl;
86     ofs << "zbounds = zlim();" << std::endl;
87     ofs << "set(gca, 'ztick', zbounds(1):" << spacing << ":zbounds"
88         "(2));" << std::endl;
89     ofs << "set(gcf, 'PaperUnits', 'inches', 'PaperPosition', [0 0
90         4 4])" << std::endl;
91     ofs << "print -dpng F:\\temp\\plot.png";
92     ofs.close();
93 }
```

```

94     ofs << "clf;" << std::endl;
95     ofs << "graphics_toolkit(\"gnuplot\")" << std::endl;
96
97     XCoordinates << "x = [";
98     YCoordinates << "y = [";
99
100    if (Samples.size() > 1)
101    {
102        Point2D s = Samples[0];
103        XCoordinates << s.GetX();
104        YCoordinates << s.GetY();
105    }
106
107    for (int i = 1; i < Samples.size(); ++i)
108    {
109        Point2D s = Samples[i];
110        XCoordinates << ", " << s.GetX();
111        YCoordinates << ", " << s.GetY();
112    }
113    XCoordinates << "];" << std::endl;
114    YCoordinates << "];" << std::endl;
115
116    ofs << XCoordinates.str() << std::endl;
117    ofs << YCoordinates.str() << std::endl;
118
119    ofs << "c = x .* y;" << std::endl;
120    ofs << "p1 = scatter(x, y, 8, c, 'filled');" << std::endl;
121    ofs << "title ('Hemisphere mapping with " << Samples.size() <<
122        " samples');" << std::endl;
123    ofs << "hold on" << std::endl;
124
125    ofs << "radius = " << radius << ";" << std::endl;
126    ofs << "theta = linspace(0, 2 * pi, 360)" << std::endl;
127    ofs << "circleX = radius*cos(theta) * 0.5;" << std::endl;
128    ofs << "circleY = radius*sin(theta) * 0.5;" << std::endl;
129    ofs << "plot( " << radius << "/2 + circleX," << radius << "/2 +
130        circleY);" << std::endl;
131
132    ofs << "grid on;" << std::endl;
133    // Sets major axis ticks to have correct spacing
134    float spacing = sqrtf(Samples.size()) / Samples.size();
135    ofs << "xbounds = xlim();" << std::endl;
136    ofs << "set(gca, 'xtick', xbounds(1):" << spacing << ":xbounds
137        (2));" << std::endl;
138    ofs << "ybounds = ylim();" << std::endl;
139    ofs << "set(gca, 'ytick', ybounds(1):" << spacing << ":ybounds
140        (2));" << std::endl;
141    ofs << "set(gcf, 'PaperUnits', 'inches', 'PaperPosition', [0 0
142        4 4])" << std::endl;
143    ofs << "print -dpng F:\\temp\\plot.png";
144    ofs.close();
145 }
```

```
141
142 protected:
143     int NumberOfSamples;
144 };
145
146
147 inline int Sampler::GetNumberOfSamples()
148 {
149     return NumberOfSamples;
150 }
```

2.1.17 Scene

Listing 2.21: Scene header file

```
1 #pragma once
2
3 #include <vector>
4
5 #include "Geometry.h"
6 #include "Primitive.h"
7 #include "Light.h"
8 #include "Camera.h"
9 #include "Integrator.h"
10 #include "Accelerators\BVH.h"
11 #include "Volume\VolumeIntegrator.h"
12
13 class Volume;
14
15 // The scene contains the <Geometry>-><Materials>, <Lights>, <Cameras>
16 class Scene
17 {
18 public:
19     Scene();
20
21     Scene(std::vector<Primitive*> geo, std::vector<Light*> lights,
22            Integrator* integrator, BVH* accelerator, Camera* camera)
23         : SGeometries(geo), SLights(lights), SIntegrator(integrator),
24           SAccelerator(accelerator), SCamera(camera),
25           SBackgroundColour(0.5f, 0.5f, 0.5f),
26           SVolumeIntegrator(nullptr), SVolumes(nullptr)
27     {
28
29     //Colour Li(const Ray& ray, Sampler* sampler, Intersection&
30     //          intersect) const
31     //{
32     //    Colour li(0, 0, 0);
33     //    float tCurr = INFINITY;
34     //    if (SAccelerator->Intersect(ray, tCurr, intersect))
35     //    {
36     //        li = SIntegrator->Li(this, ray, sampler, intersect);
37     //    }
38     //    else
39     //    {
40     //        return Colour(0.5, 0.5, 0.5);
41     //    }
42     //}
43
44     bool Intersect(const Ray& ray, Intersection& intersection) const
45     {
```

```

46         float tmin = INFINITY;
47         return SAccelerator->Intersect(ray, tmin, intersection);
48     }
49
50     const BBox& GetBoundingBox() const
51     {
52         return SAccelerator->GetBoundingBox();
53     }
54
55     void AddGeometry(Primitive* geometry);
56     void AddLight(Light* light);
57     const Camera* GetCamera() const;
58     void SetCamera(Camera* camera);
59     std::vector<Light*> GetLights() const { return SLights; }
60     std::vector<Primitive*> GetSpecularGeometry() const { return
61         SSpecularGeometries; }
62     BVH* GetAccelerator() const { return SAccelerator; }
63     Colour GetBackgroundColour() const { return SBackgroundColour; }
64
65     Volume* GetVolume() const;
66     std::vector<Primitive*> SSpecularGeometries;
67     VolumeIntegrator* SVolumeIntegrator;
68     Volume* SVolumes;
69
70     private:
71         std::vector<Primitive*> SGeometries;
72         std::vector<Light*> SLights;
73         Camera* SCamera;
74         Integrator* SIntegrator;
75         Colour SBackgroundColour;
76         BVH* SAccelerator;
77     };

```

Listing 2.22: Scene source file

```
1 #include "Scene.h"
2
3 Scene::Scene()
4     : SBackgroundColour{ 0, 0, 0 }, SGeometries{}, SLights{}, SCamera{
5         nullptr,
6     SVolumeIntegrator(nullptr), SVolumes(nullptr)
7 }
8
9 void Scene::AddGeometry(Primitive* geometry)
10 {
11     SSpecularGeometries.push_back(geometry);
12 }
13 void Scene::AddLight(Light* light)
14 {
15     SLights.push_back(light);
16 }
17
18
19 const Camera* Scene::GetCamera() const
20 {
21     return SCamera;
22 }
23
24 void Scene::SetCamera(Camera* camera)
25 {
26     SCamera = camera;
27 }
28
29 Volume* Scene::GetVolume() const
30 {
31     return SVolumes;
32 }
```

2.1.18 Texture

Listing 2.23: Texture header file

```
1 #pragma once
2 #include "Colour.h"
3 #include "../Core/Math/Point.h"
4 #include "Intersection.h"
5
6 class Texture
7 {
8 public:
9     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
10    const = 0;
11 };
```

2.2 Accelerators

2.2.1 BVH

Listing 2.24: BVH header file

```
1 #pragma once
2
3 #include "../Primitive.h"
4 #include "../Ray.h"
5 #include <vector>
6 #include <algorithm>
7 #include <functional>
8 #include "../../Core/Utility/Timer.h"
9 using namespace std;
10
11 class BVH : public Primitive
12 {
13 public:
14
15     struct BVHCompare
16     {
17         int axis;
18         BVHCompare(){}
19         BVHCompare(int axis)
20             :axis(axis){}
21
22         bool operator()(Primitive* a, Primitive* b)
23         {
24             return a->GetBounds().GetCenter()[axis] < b->GetBounds().
25                 GetCenter()[axis];
26         }
27     };
28
29     struct CentroidCompare
30     {
31         Point& centroid;
32         int axis;
33         CentroidCompare(int a, Point& c)
34             :axis(a), centroid(c)
35         {}
36
37         bool operator()(Primitive* a)
38         {
39             return a->GetBounds().GetCenter()[axis] < centroid[axis];
40         }
41     };
42
43     struct BVHNode
44     {
45         BBox Volume;
46         BVHNode* left = nullptr;
```

```

46     BVHNode* right = nullptr;
47     vector<Primitive*> objects;
48
49     bool IsLeaf()
50     {
51         return (left == nullptr && right == nullptr);
52     }
53 };
54
55 BVH(Primitive* primitive)
56     :Root(new BVHNode()), Compare(new BVHCompare())
57 {
58     if (primitive->IsCompound())
59     {
60         std::vector<Primitive*> primitives = primitive->
61             GetPrimitive();
62         Objects.insert(Objects.end(), primitives.begin(),
63                         primitives.end());
64     }
65     else
66     {
67         Objects.push_back(primitive);
68     }
69 }
70
71 BVH(std::vector<Primitive*> primitives)
72     :Root(new BVHNode()), Compare(new BVHCompare())
73 {
74     for (Primitive* primitive : primitives)
75     {
76         if (primitive->IsCompound())
77         {
78             std::vector<Primitive*> primitives = primitive->
79                 GetPrimitive();
80             Objects.insert(Objects.end(), primitives.begin(),
81                           primitives.end());
82         }
83         else
84         {
85             Objects.push_back(primitive);
86         }
87     }
88 }
89
90 void Build(BVHNode* node, vector<Primitive*> objects)
91 {
92     BBox volume;
93     for (int i = 0; i < objects.size(); ++i)

```

```

94     {
95         volume = volume + objects[i]->GetBounds();
96     }
97     node->Volume = volume;
98     node->objects = objects;
99
100    if (objects.size() < 6)
101    {
102        return;
103    }
104
105    int longestAxis = volume.LongestAxis();
106    Point centroid = volume.GetCenter();
107    //auto it = std::partition(objects.begin(), objects.end(),
108    //    CentroidCompare(longestAxis, centroid));
109    //vector<Geometry*> left(objects.begin(), it);
110    //vector<Geometry*> right(it, objects.end());
111    //if (left.size() == objects.size() || right.size() == objects.
112    //    size())
113    //    return;
114
115    Compare->axis = longestAxis;
116    std::sort(objects.begin(), objects.end(), *Compare);
117    vector<Primitive*> left(objects.begin(), objects.end() -
118        objects.size() / 2);
119    vector<Primitive*> right(objects.begin() + objects.size() / 2,
120        objects.end());
121    node->left = new BVHNode();
122    node->right = new BVHNode();
123
124    Build(node->left, left);
125    Build(node->right, right);
126}
127
128 bool Intersect(const Ray& ray, float& tCurr, Intersection& is)
129 const
130 {
131     return Intersect(Root, ray, tCurr, is);
132 }
133
134 bool Intersect(const Ray& ray, float& tCurr, Intersection& is, bool
135 shadow) const
136 {
137     return Intersect(Root, ray, tCurr, is, shadow);
138 }
139
140 bool Intersect(BVHNode* node, const Ray& ray, float& tCurr,
141 Intersection& is) const
142 {
143     // Epsilon should be scene dependent. "Holy" mesh otherwise.
144     float epsilon = 1e-6;
145     float tMin = INFINITY;

```

```

139     bool hit = false;
140     if (node->Volume.IntersectRay(ray, tCurr, epsilon, is))
141     {
142         if (node->IsLeaf())
143         {
144
145             for (Primitive* geo : node->objects)
146             {
147                 if (geo->IntersectRay(ray, tCurr, epsilon, is) &&
148                     tCurr < tMin)
149                 {
150                     hit = true;
151                     tMin = tCurr;
152                 }
153             }
154
155             return hit;
156         }
157         bool hit1 = false, hit2 = false;
158
159         if (Intersect(node->left, ray, tCurr, is) && tCurr < tMin)
160         {
161             tMin = tCurr;
162             hit1 = true;
163         }
164         if (Intersect(node->right, ray, tCurr, is) && tCurr < tMin)
165         {
166             tMin = tCurr;
167             hit2 = true;
168         }
169         return hit1 || hit2;
170     }
171
172     return false;
173 }
174
175
176     bool Intersect(BVHNode* node, const Ray& ray, float& tCurr,
177                   Intersection& is, bool shadow) const
178     {
179         float epsilon = 1e-6;
180         float tMin = INFINITY;
181         bool hit = false;
182         if (node->Volume.IntersectRay(ray, tCurr, epsilon, is))
183         {
184             if (node->IsLeaf())
185             {
186                 for (Primitive* geo : node->objects)
187                 {

```

```

188         if (geo->IntersectRay(ray, tCurr, epsilon, is) &&
189             tCurr < tMin)
190             {
191                 return true;
192             }
193         }
194         return hit;
195     }
196     bool hit1 = false, hit2 = false;
197
198     if (Intersect(node->left, ray, tCurr, is) && tCurr < tMin)
199     {
200         tMin = tCurr;
201         hit1 = true;
202     }
203     if (Intersect(node->right, ray, tCurr, is) && tCurr < tMin)
204     {
205         tMin = tCurr;
206         hit2 = true;
207     }
208     return hit1 || hit2;
209
210 }
211
212     return false;
213 }
214
215 const BBox& GetBoundingBox() const
216 {
217     return Root->Volume;
218 }
219
220
221 Material* GetMaterial() const override;
222 BBox GetBounds() const override;
223 bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
224                   Intersection& intersection) const override;
225 std::vector<Primitive*> GetPrimitive() const override;
226
227     vector<Primitive*> Objects;
228     BVHCompare* Compare;
229     BVHNode* Root;
230 };
231 inline Material* BVH::GetMaterial() const
232 {
233     std::set_unexpected([]{
234         std::cerr << "Call to GetMaterial From BVH\n";
235         std::abort();
236     });

```

```
237     return nullptr;
238 }
239
240 inline BBox BVH::GetBounds() const
241 {
242     return Root->Volume;
243 }
244
245 inline bool BVH::IntersectRay(const Ray& ray, float& tMin, float&
246     epsilon, Intersection& intersection) const
247 {
248     return Intersect(ray, tMin, intersection);
249 }
250
251 inline std::vector<Primitive*> BVH::GetPrimitive() const
252 {
253     std::set_unexpected([]{
254         std::cerr << "Call to GetPrimitive From BVH\n";
255         std::abort();
256     });
257     return {};
258 }
```

2.3 BRDF

2.3.1 BlinnMicrofacetBRDF

Listing 2.25: BlinnMicrofacetBRDF header file

```
1 #pragma once
2 #include "MicrofacetBRDF.h"
3 #include "Dielectric.h"
4 //class BlinnDistribution : public MicrofacetDistribution
5 //{
6 //public:
7 //    float Pdf(const Vector& wo, const Vector& wi, Intersection&
8 //              intersection) const override;
9 //    BlinnDistribution(float roughness);
10 //    virtual float D(const Vector& wh) const override;
11 //    virtual void SampleF(const Vector& wo, Vector& wi, float& pdf,
12 //                          Intersection& intersection) const override;
13 //
14 //private:
15 //    float alpha;
16 //};
17 //inline float BlinnDistribution::Pdf(const Vector& wo, const Vector&
18 //                                    wi, Intersection& intersection) const
19 //{
20 //
21 //inline BlinnDistribution::BlinnDistribution(float roughness)
22 //    : alpha(roughness * roughness)
23 //{
24 //
25 //}
26 //
27 //inline void BlinnDistribution::SampleF(const Vector& wo, Vector& wi,
28 //                                       float& pdf, Intersection& intersection) const
29 //{
30 //    Point2D sample(rand() / float(RAND_MAX), rand() / float(RAND_MAX));
31 //    wi = Sample::Blinn(sample, alpha);
32 //    pdf =
33 //}
34 //inline float BlinnDistribution::D(const Vector& wh) const
35 //{
36 //    return (1.0f / (M_PI * alpha * alpha)) * powf(OrthonormalBasis::
37 //        CosTheta(wh), (2.0f / (alpha * alpha) - 2));
38 //}
39 class BlinnMicrofacetBRDF : public MicrofacetBRDF
40 {
41 public:
```

```

42     BlinnMicrofacetBRDF(float alpha, Texture* diffuseTexture, float ior
43     , Texture* roughnessMap = nullptr)
44     : MicrofacetBRDF(BRDF::Type::GLOSSY, ior, new Dielectric(1.0029
45         f, ior)),
46         alpha(alpha),
47         IOR(ior),
48         kd(diffuseTexture),
49         RoughnessMap(roughnessMap)
50     {
51
52     Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
53         Intersection& intersection) const override;
54     float PDF(const Vector& wo, const Vector& wi, Intersection&
55         intersection) const override;
56     float D(const Vector& wh, Intersection& intersection) const
57         override;
58
59     private:
60         float alpha;
61         float IOR;
62         Texture* kd;
63         Texture* RoughnessMap;
64     };
65
66     inline Colour BlinnMicrofacetBRDF::SampleF(const Vector& wo, Vector& wi
67     , float& pdf, Intersection& intersection) const
68     {
69         Point2D sample(rand() / float(RAND_MAX), rand() / float(RAND_MAX));
70         float phi = 2 * M_PI * sample[1];
71         float roughness = alpha;
72         if (RoughnessMap != nullptr)
73         {
74             roughness = (RoughnessMap->GetValue(intersection.U,
75                 intersection.V, intersection)[0]) * 10000;
76         }
77
78         float costheta = powf(sample[0], 1.0f / (roughness + 1));
79         float sintheta = sqrtf(std::max(0.0f, 1 - costheta * costheta));
80         Vector wh = Vector(sintheta * sin(phi), sintheta * cos(phi),
81             costheta);
82         if (!IsSameHemisphere(Vector(wo), wh))
83         {
84             wh = -wh;
85         }
86         wi = -wo + 2.0f * Dot(wo, wh) * wh;
87         pdf = ((roughness + 1) * powf(costheta, roughness)) / (2 * M_PI * 4
88             * Dot(wo, wh));
89         if (Dot(wo, wh) < 0.0f)
90             pdf = 0;
91
92         if (kd != nullptr)

```

```

85         return F(wo, wi, intersection) * kd->GetValue(intersection.U,
86             intersection.V, intersection);
87     }
88
89     inline float BlinnMicrofacetBRDF::PDF(const Vector& wo, const Vector&
90         wi, Intersection& intersection) const
91     {
92         Vector wh = (wo + wi).Hat();
93         float costheta = OrthonormalBasis::CosTheta(wh);
94         float roughness = alpha;
95         if (RoughnessMap != nullptr)
96             roughness = (RoughnessMap->GetValue(intersection.U,
97                 intersection.V, intersection)[0]) * 10000;
98         float pdf = ((roughness + 1) * powf(costheta, roughness)) / (2.0f *
99             M_PI * 4.0f * Dot(wo, wh));
100        if (Dot(wo, wh) <= 0.0f)
101        {
102            pdf = 0.0f;
103        }
104    }
105
106    inline float BlinnMicrofacetBRDF::D(const Vector& wh, Intersection&
107        intersection) const
108    {
109        float costheta = OrthonormalBasis::AbsCosTheta(wh);
110        float roughness = alpha;
111        if (RoughnessMap != nullptr)
112        {
113            roughness = (RoughnessMap->GetValue(intersection.U,
114                 intersection.V, intersection)[0]) * 10000;
115        }
116        if (roughness == 0)
117        {
118            return 0.0f;
119        }
120        return (roughness + 2) * (1 / (2 * M_PI)) * powf(costheta,
121             roughness);
122    }

```

2.3.2 Conductor

Listing 2.26: Conductor header file

```
1 #pragma once
2 #include "Fresnel.h"
3
4 class Conductor : public Fresnel
5 {
6 public:
7
8     Conductor(float eta, float k)
9         : Eta(eta),
10           K(k)
11     {
12     }
13
14     Colour Evaluate(float cosTheta_i) const override;
15 private:
16     float Eta;
17     float K;
18 };
19
20 inline Colour Conductor::Evaluate(float cosTheta_i) const
21 {
22     return Fresnel::Conductor(fabsf(cosTheta_i), Eta, K);
23 }
```

2.3.3 Dielectric

Listing 2.27: Dielectric header file

```
1 #pragma once
2 #include "Fresnel.h"
3
4 class Dielectric : public Fresnel
5 {
6 public:
7     Dielectric(float eta_i, float eta_t)
8         : EtaI(eta_i),
9          EtaT(eta_t)
10    { }
11
12     Colour Evaluate(float cosTheta_i) const override;
13 private:
14     float EtaI;
15     float EtaT;
16 };
17
18 inline Colour Dielectric::Evaluate(float cosTheta_i) const
19 {
20     cosTheta_i = clamp(cosTheta_i, -1.0f, 1.0f);
21
22     float eta_i = EtaI, eta_t = EtaT;
23     bool entering = cosTheta_i > 0.0f;
24     if (!entering)
25     {
26         std::swap(eta_i, eta_t);
27     }
28
29     float sinTheta_t = (EtaI / EtaT) * sqrtf(Max(0.0f, 1.0f -
30         cosTheta_i * cosTheta_i));
31     // total internal reflection
32     if (sinTheta_t >= 1.0f)
33     {
34         return Colour(1.0f);
35     }
36     float cosTheta_t = sqrtf(Max(0.0f, 1 - sinTheta_t * sinTheta_t));
37     return Fresnel::Dielectric(fabsf(cosTheta_i), cosTheta_t, EtaI,
38                                EtaT);
39 }
```

2.3.4 DielectricBSDF

Listing 2.28: DielectricBSDF header file

```
1 #pragma once
2 #include "Dielectric.h"
3 #include "../BRDF.h"
4 #include "../../Core/Math/OrthonormalBasis.h"
5
6 class DielectricBSDF : public BRDF
7 {
8 public:
9     DielectricBSDF(const Colour& transmission_k, float eta_i, float
10                 eta_t)
11         : BRDF(BRDF::Type::REFRACTION),
12           TransmissionK(transmission_k),
13           EtaI(eta_i),
14           EtaT(eta_t),
15           dielectric_(new Dielectric(eta_i, eta_t))
16     {}
17
18     Colour F(const Vector& wo, const Vector& wi, Intersection&
19               intersection) const override;
20     Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
21                      Intersection& intersection) const override;
22     Colour Rho(const Vector& wo, Intersection& intersection) const
23                     override;
24
25 private:
26     Colour TransmissionK;
27     float EtaI;
28     float EtaT;
29     Dielectric* dielectric_;
30 };
31
32 inline Colour DielectricBSDF::F(const Vector& wo, const Vector& wi,
33                                  Intersection& intersection) const
34 {
35     return Colour(0.0f);
36 }
37
38 // See page 443 PBRT Specular Transmission
39 inline Colour DielectricBSDF::SampleF(const Vector& wo, Vector& wi,
40                                       float& pdf, Intersection& intersection) const
41 {
42     pdf = 1.0f;
43     float etai = EtaI;
44     float etat = EtaT;
45     // Since we are in local space the cosine points in the direction
46     // of the normal
47     bool isEntering = OrthonormalBasis::CosTheta(wo) > 0.0f;
48     float eta = EtaI / EtaT;
```

```

43     if (!isEntering)
44     {
45         eta = 1 / eta;
46     }
47
48     // Use Snell's law to compute transmitted direction
49     float sinI2 = OrthonormalBasis::SinTheta2(wo);
50
51     float sinT2 = eta * eta * sinI2;
52     if (sinT2 >= 1.0f)
53     {
54         //return Colour(1.0f, 1.0f, 1.0f);
55         pdf = 1.0f;
56         //wi = 2 * Dot(wo, intersection.GetNormal()) * Vector(
57             //intersection.GetNormal()) - wo;
58         wi = Vector(-wo.GetX(), -wo.GetY(), wo.GetZ()).Hat();
59         return 1 / OrthonormalBasis::AbsCosTheta(wi);
60     }
61     float cost = sqrtf(Max(0.0f, 1.0f - sinT2));
62     if (isEntering)
63         cost = -cost;
64     wi = Vector(eta * -wo.GetX(), eta * -wo.GetY(), cost).Hat();
65     // Evaluate the amount of transmission using the fresnel equation
66     Colour F = dielectric_->Evaluate(OrthonormalBasis::CosTheta(wo));
67     Colour transmissionAmount = (Colour(1.0f) - F);
68     return transmissionAmount / OrthonormalBasis::AbsCosTheta(wi);
69 }
70
71 inline Colour DielectricBSDF::Rho(const Vector& wo, Intersection&
72     intersection) const
73 {
74     return Colour(1) - dielectric_->Evaluate(OrthonormalBasis::CosTheta
75         (wo));
76 }
```

2.3.5 Fresnel

Listing 2.29: Fresnel header file

```
1 #pragma once
2
3 #include "../Colour.h"
4
5 class Fresnel
6 {
7 protected:
8     ~Fresnel()
9     {
10 }
11
12 public:
13
14     static Colour Dielectric(float cosTheta_i, float cosTheta_t, float
15         eta_i, float eta_t);
16     static Colour Conductor(float cosTheta_i, float eta_i, float
17         absorptionCoeffK);
18     virtual Colour Evaluate(float cosTheta_i) const = 0;
19 };
20
21 class SchlickFresnel : public Fresnel
22 {
23 public:
24     SchlickFresnel(float ior);
25     Colour Evaluate(float cosTheta_i) const override;
26 private:
27     float IOR;
28 };
29
30 inline SchlickFresnel::SchlickFresnel(float ior)
31     : IOR(ior)
32 {
33 }
34
35 inline Colour SchlickFresnel::Evaluate(float cosTheta_i) const
36 {
37     // reflectance at normal incidence
38     Colour Ro = ((1 - IOR) / (1 + IOR));
39     Ro = Ro * Ro;
40     return Ro + (Colour(1.0f) - Ro) * powf(1 - cosTheta_i, 5.0f);
41 }
42
43 inline Colour Fresnel::Dielectric(float cosTheta_i, float cosTheta_t,
44         float eta_i, float eta_t)
45 {
46     Colour ReflectanceParallel = (eta_t * cosTheta_i - eta_i *
47         cosTheta_t) /
```

```

45                         (eta_t * cosTheta_i + eta_i *
46                          cosTheta_t);
46     Colour ReflectancePerpendicular = (eta_i * cosTheta_i - eta_t *
47                                         cosTheta_t) /
47                                         (eta_i * cosTheta_i + eta_t *
48                                         cosTheta_t);
48     return 0.5f * (ReflectanceParallel * ReflectanceParallel +
49                     ReflectancePerpendicular * ReflectancePerpendicular);
49 }
50
51 inline Colour Fresnel::Conductor(float cosTheta_i, float eta_i, float
52 absorptionCoeffK)
52 {
53     Colour A = (eta_i * eta_i + absorptionCoeffK * absorptionCoeffK);
54     Colour B = A * cosTheta_i * cosTheta_i;
55     Colour ReflectanceParallel = (B - (2.0f * eta_i * cosTheta_i) + 1)
56     /
56             (B + (2.0f * cosTheta_i) + 1);
57     Colour ReflectancePerpendicular = (A - (2.0f * eta_i * cosTheta_i)
58                                         + cosTheta_i * cosTheta_i) /
58                                         (A + (2.0f * eta_i * cosTheta_i)
59                                         + cosTheta_i * cosTheta_i);
59     return 0.5f * (ReflectanceParallel + ReflectancePerpendicular);
60 }
```

2.3.6 GlossySpecular

Listing 2.30: GlossySpecular header file

```
1 #pragma once
2
3 #include "../BRDF.h"
4 class GlossySpecular : public BRDF
5 {
6 public:
7     GlossySpecular(Colour& ca)
8         : BRDF(BRDF::Type::GLOSSY), Specular{ ca }
9     {}
10
11     virtual Colour F(const Vector& wo, const Vector& wi, Intersection&
12                     intersection) const override
13     {
14         return Colour(0.0f, 0.0f, 0.0f);
15         //return Specular;
16     }
17
18     virtual Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
19                            Intersection& intersection) const override
20     {
21
22         //wi = Vector(-wo);
23         //return Specular;
24         return Colour(0.0f, 0.0f, 0.0f);
25     }
26
27     virtual Colour Rho(const Vector& wo, Intersection& intersection)
28                     const override
29     {
30         return Colour(0, 0, 0);
31     }
32
33 private:
34     Colour Specular;
35 };
```

2.3.7 Lambertian

Listing 2.31: Lambertian header file

```
1 #pragma once
2
3 #include "../BRDF.h"
4 #include "../Sampler/RandomSampler.h"
5 #include "../Colour.h"
6 #include "../Texture.h"
7
8 class Lambertian : public BRDF
9 {
10 public:
11     Lambertian(Texture* ca)
12         : BRDF(Type::DIFFUSE), DiffuseAlbedo(ca)
13     {}
14
15     virtual Colour F(const Vector& wo, const Vector& wi, Intersection&
16                       intersection) const override
17     {
18         return DiffuseAlbedo->GetValue(intersection.U, intersection.V,
19                                         intersection) * M_INVPI;
20     }
21
22     virtual Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
23                            Intersection& intersection) const override
24     {
25         //return BRDF::SampleF(wo, wi, pdf, intersection);
26         Point2D sample(rand() / float(RAND_MAX), rand() / float(
27             RAND_MAX));
28         wi = Sample::CosineHemisphere(sample);
29         pdf = Sample::CosineHemispherePDF(OrthonormalBasis::CosTheta(wi
30             ));
31         if (pdf <= 0.0f)
32         {
33             pdf = 0;
34             return Colour(0.0f);
35         }
36         return F(wo, wi, intersection);
37     }
38
39     virtual Colour Rho(const Vector& wo, Intersection& intersection)
40                     const override
41     {
42         return DiffuseAlbedo->GetValue(intersection.U, intersection.V,
43                                         intersection);
44     }
45
46 private:
47     Texture* DiffuseAlbedo;
48 };
```

2.3.8 MicrofacetBRDF

Listing 2.32: MicrofacetBRDF header file

```
1 #pragma once
2 #include "../BRDF.h"
3 #include "Fresnel.h"
4 #include "../../Core/Math/OrthonormalBasis.h"
5
6 class MicrofacetDistribution
7 {
8 public:
9     virtual float D(const Vector& wh, float alpha) const = 0;
10    virtual float G1(const Vector& v, const Vector& m, float alpha)
11        const = 0;
12    virtual float G(const Vector& wo, const Vector& wi, const Vector& m
13        , float alpha) const;
14    //virtual float PDF(const Vector& wo, const Vector& wi, float alpha
15        );
16    virtual Vector Sample(const Point2D& sample, float alpha) const =
17        0;
18 };
19
20 class PhongDistribution : public MicrofacetDistribution
21 {
22 public:
23     float D(const Vector& wh, float alpha) const override;
24     float G1(const Vector& v, const Vector& m, float alpha) const
25         override;
26     //float PDF(const Vector& wo, const Vector& wi, float alpha)
27         override;
28     Vector Sample(const Point2D& sample, float alpha) const override;
29 };
30
31 inline float MicrofacetDistribution::G(const Vector& wo, const Vector&
32     wi, const Vector& m, float alpha) const
33 {
34     return G1(wi, m, alpha) * G1(wo, m, alpha);
35 }
36
37 //inline float MicrofacetDistribution::PDF(const Vector& wo, const
38 //    Vector& wi, float alpha)
39 //{
40 //    Vector wh = (wo + wi).Hat();
41 //    return D(wh, alpha) / fabsf(OrthonormalBasis::CosTheta(wh));
42 //}
43
44 inline Vector PhongDistribution::Sample(const Point2D& sample, float
45     alpha) const
46 {
47     return Sample::Phong(sample, alpha);
48 }
```

```

41 inline float PhongDistribution::D(const Vector& wh, float alpha) const
42 {
43     return Sample::PhongPdf(wh, alpha);
44 }
45
46 inline float PhongDistribution::G1(const Vector& v, const Vector& m,
47     float alpha) const
48 {
49     if (Dot(v, m) > 0.0f)
50     {
51         if (alpha < 1.6)
52         {
53             float alpha2 = alpha * alpha;
54             return (3.535 * alpha + 2.181 * alpha2) / (1 + 2.276 * alpha +
55                 2.577 * alpha2);
56         }
57     }
58     return 1.0f;
59 }
60
61 class BeckmannDistribution : public MicrofacetDistribution
62 {
63 public:
64     float D(const Vector& wh, float alpha) const override;
65     float G1(const Vector& v, const Vector& m, float alpha) const
66         override;
67     //float PDF(const Vector& wo, const Vector& wi, float alpha)
68     //    override;
69     Vector Sample(const Point2D& sample, float alpha) const override;
70 };
71
72 inline float BeckmannDistribution::D(const Vector& wh, float alpha)
73     const
74 {
75     alpha = 1 - alpha;
76     return Sample::BeckmannPdf(wh, alpha);
77 }
78
79 inline float BeckmannDistribution::G1(const Vector& v, const Vector& m,
80     float alpha) const
81 {
82     alpha = 1 - alpha;
83     if (Dot(v, m) / OrthonormalBasis::CosTheta(v) > 0.0f)
84     {
85         if (alpha < 1.6f)
86         {
87             float alpha2 = alpha * alpha;
88             return (3.535 * alpha + 2.181 * alpha2) / (1 + 2.276 *
89                 alpha + 2.577 * alpha2);

```

```

86         }
87     }
88     return 1.0f;
89 }
90
91 inline Vector BeckmannDistribution::Sample(const Point2D& sample, float
92     alpha) const
93 {
94     alpha = 1 - alpha;
95     return Sample::Beckmann(sample, alpha);
96 }
97
98
99 class GGXDistribution : public MicrofacetDistribution
100 {
101 public:
102     float D(const Vector& wh, float alpha) const override;
103     float G1(const Vector& v, const Vector& m, float alpha) const
104         override;
105     //float PDF(const Vector& wo, const Vector& wi, float alpha)
106         override;
107     Vector Sample(const Point2D& sample, float alpha) const override;
108 };
109
110 inline float GGXDistribution::D(const Vector& wh, float alpha) const
111 {
112     return Sample::GGXPdf(wh, alpha);
113 }
114
115 inline float GGXDistribution::G1(const Vector& v, const Vector& m,
116     float alpha) const
117 {
118     if (Dot(v, m) > 0.0f)
119     {
120         float costheta = v[2];
121         float alpha2 = alpha * alpha;
122         float cosTheta2 = costheta * costheta;
123         float tanTheta2 = Max(0.0f, 1.0f - cosTheta2) / cosTheta2;
124         return 2.0f / (1.0f + sqrt(1.0f + alpha2*tanTheta2));
125     }
126     return 0.0f;
127 }
128
129 inline Vector GGXDistribution::Sample(const Point2D& sample, float
130     alpha) const
131 {
132     return Sample::GGX(sample, alpha);

```

```

133
134 class MicrofacetBRDF : public BRDF
135 {
136 public:
137     MicrofacetBRDF(BRDF::Type type, float ior, Fresnel* fresnel =
138                     nullptr);
139
140     Colour F(const Vector& wo, const Vector& wi, Intersection&
141               intersection) const override;
142
143     Colour Rho(const Vector& wo, Intersection& intersection) const
144               override;
145
146     virtual Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
147                            Intersection& intersection) const override = 0;
148
149     virtual float PDF(const Vector& wo, const Vector& wi, Intersection&
150                       intersection) const override;
151
152     virtual float D(const Vector& wh, Intersection& intersection) const
153             = 0;
154
155     virtual float G(const Vector& wo, const Vector& wi, const Vector&
156                      wh, Intersection& intersection) const;
157
158     inline MicrofacetBRDF::MicrofacetBRDF(BRDF::Type type, float ior,
159                                             Fresnel* _fresnel)
160         : BRDF(type), fresnel(_fresnel)
161     {
162         if (fresnel == nullptr)
163         {
164             fresnel = new SchlickFresnel(ior);
165         }
166     }
167
168     inline Colour MicrofacetBRDF::F(const Vector& wo, const Vector& wi,
169                                     Intersection& intersection) const
170     {
171         Vector wh = (wo + wi).Hat();
172         //Colour fresnel = Colour(1.0f);
173         //Colour Rs = Colour(1.0f);
174         //float ior = 1.5;
175         //Rs = ((1 - ior) / (1 + ior));
176         //Rs = Rs * Rs;
177         //float costheta = Dot(wh, wi);

```

```

176     //Colour fresnel = Rs + (Colour(1.0f) - Rs) * powf(1 - costheta,
177     //      5.0f);
178     float costheta = Dot(wh, wi);
179     Colour F = fresnel->Evaluate(costheta);
180     return (D(wh, intersection) * G(wo, wi, wh, intersection) * F) /
181           (4.0f * OrthonormalBasis::CosTheta(wo) * OrthonormalBasis::
182           CosTheta(wi));
183 }
184
185
186
187 inline float MicrofacetBRDF::Rho(const Vector& wo, Intersection&
188     intersection) const
189 {
190     return 0.0f;
191 }
192
193 inline float MicrofacetBRDF::PDF(const Vector& wo, const Vector& wi,
194     Intersection& intersection) const
195 {
196     // Cook-Torrance Geometric Term
197     float nh = OrthonormalBasis::CosTheta(wh);
198     float nv = OrthonormalBasis::CosTheta(wo);
199     float nl = OrthonormalBasis::CosTheta(wi);
200     float vh = Dot(wo, wh);
201
202     float term1 = (2.0f * nh * nv) / vh;
203     float term2 = (2.0f * nh * nl) / vh;
204     return Min(1.0f, Min(term1, term2));
205 }
```

2.3.9 SpecularReflection

Listing 2.33: SpecularReflection header file

```
1 #pragma once
2
3 #include "../BRDF.h"
4 #include "Fresnel.h"
5 #include "../../Core/Math/OrthonormalBasis.h"
6
7 class MirrorReflection : public BRDF
8 {
9 public:
10     MirrorReflection(Colour& ca, Fresnel* fresnel, Sampler* sampler)
11         : BRDF(BRDF::Type::SPECULAR), Specular{ ca }, fresnel(fresnel)
12     {}
13
14     virtual Colour F(const Vector& wo, const Vector& wi, Intersection&
15                     intersection) const override
16     {
17         return Colour(0, 0, 0);
18     }
19
20     virtual Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
21                            Intersection& intersection) const override
22     {
23         pdf = 1.0f;
24         wi = Vector(-wo.GetX(), -wo.GetY(), wo.GetZ()).Hat();
25         return fresnel->Evaluate(OrthonormalBasis::CosTheta(wo)) *
26             Specular / OrthonormalBasis::AbsCosTheta(wi);
27     }
28
29     virtual Colour Rho(const Vector& wo, Intersection& intersection)
30                     const override
31     {
32         return Colour(0, 0, 0);
33     }
34 }
```

2.3.10 WardBRDF

Listing 2.34: WardBRDF header file

```
1 #pragma once
2
3 #include "../Texture.h"
4 #include <math.h>
5 #include "MicrofacetBRDF.h"
6 class WardBRDF : public BRDF
7 {
8 public:
9
10    WardBRDF(Colour& ks, float ax, float ay, Texture* anisotropyMap =
11               nullptr)
12        : BRDF(BRDF::Type::GLOSSY),
13          ks(ks), ax(ax), ay(ay), AnisotropyMap(anisotropyMap)
14    {
15        ax = clamp(ax, 0.001f, 1.0f);
16        ay = clamp(ay, 0.001f, 1.0f);
17    }
18
19    virtual Colour Rho(const Vector& wo, Intersection& intersection)
20        const override;
21    Colour SampleF(const Vector& wo, Vector& wi, float& pdf,
22                    Intersection& intersection) const override;
23    inline Colour F(const Vector& wo, const Vector& wi, Intersection&
24                     intersection) const override;
25    float PDF(const Vector& wo, const Vector& wi, Intersection&
26               intersection) const override;
27 private:
28    Colour ks;
29    float ax, ay;
30    Texture* AnisotropyMap;
31    PhongDistribution phong;
32 };
33
34 inline Colour WardBRDF::Rho(const Vector& wo, Intersection&
35                               intersection) const
36 {
37     return 0.0f;
38 }
39
40 inline Colour WardBRDF::SampleF(const Vector& wo, Vector& wi, float&
41                                   pdf, Intersection& intersection) const
42 {
43     Point2D sample(rand() / float(RAND_MAX), rand() / float(RAND_MAX));
44     float nx = ax;
45     float ny = ay;
46     if (AnisotropyMap)
47     {
48         float tempX = AnisotropyMap->GetValue(intersection.U,
49                                         intersection.V, intersection)[0];
```

```

42     ny = tempX;
43     nx = (1 - ny);
44     nx = clamp(nx, 0.001f, 1.0f);
45     ny = clamp(ny, 0.001f, 1.0f);
46 }
47 float phi = atanf((ny / nx) * tanf(2 * M_PI * sample[1]));
48 if (sample[1] <= 0.25 || sample[1] >= 0.75)
49 {
50     phi += M_PI;
51 }
52 float cosPhi = cosf(phi);
53 float sinPhi = sinf(phi);
54 float theta = atanf(sqrtf(-logf(sample[0]) / (cosPhi * cosPhi / (nx
55     * nx) + sinPhi * sinPhi / (ny * ny))));
56 float costheta = cosf(theta);
57 float sintheta = sinf(theta);
58
59 Vector wh = Vector(sintheta * cosPhi, sintheta * sinPhi, costheta);
60 if (Dot(wh, wo) < 0.0f)
61 {
62     wh = -wh;
63 }
64 wi = -wo + 2.0f * Dot(wo, wh) * wh;
65 pdf = PDF(wo, wi, intersection);
66 return F(wo, wi, intersection);
67 }
68
69 inline Colour WardBRDF::F(const Vector& wo, const Vector& wi,
70 Intersection& intersection) const
71 {
72     float nx = ax;
73     float ny = ay;
74     if (AnisotropyMap)
75     {
76         float tempX = AnisotropyMap->GetValue(intersection.U,
77             intersection.V, intersection)[0];
78         ny = tempX;
79         nx = (1 - ny);
80         nx = clamp(nx, 0.001f, 1.0f);
81         ny = clamp(ny, 0.001f, 1.0f);
82     }
83     Vector wh = (wo + wi).Hat();
84     float win = OrthonormalBasis::AbsCosTheta(wi);
85     float won = OrthonormalBasis::AbsCosTheta(wo);
86     float hn = OrthonormalBasis::CosTheta(wh);
87     hn = acosf(hn);
88     Colour a = ks / (4 * M_PI * nx * ny * sqrtf(win * won));
89     float b = -tanf(hn) * tanf(hn) * (OrthonormalBasis::CosPhi(wh) *
90         OrthonormalBasis::CosPhi(wh) / (nx * nx) +

```

```

89         OrthonormalBasis::SinPhi(wh) * OrthonormalBasis::SinPhi(wh) / (
90             ny * ny));
90     Colour c = a * expf(b);
91     return c;
92 }
93 }
94
95 inline float WardBRDF::PDF(const Vector& wo, const Vector& wi,
96     Intersection& intersection) const
96 {
97     float nx = ax;
98     float ny = ay;
99     if (AnisotropyMap)
100    {
101        float tempX = AnisotropyMap->GetValue(intersection.U,
102            intersection.V, intersection)[0];
102        ny = tempX;
103        nx = (1 - ny);
104        nx = clamp(nx, 0.001f, 1.0f);
105        ny = clamp(ny, 0.001f, 1.0f);
106    }
107
108    Vector wh = (wo + wi).Hat();
109    float hi = Dot(wh, wo);
110    float cosTheta2 = OrthonormalBasis::CosTheta(wh) * OrthonormalBasis
111        ::CosTheta(wh);
112    float tanTheta2 = (1 - cosTheta2) / cosTheta2;
113
114    float a = 1 / (4 * M_PI * nx * ny * hi * powf(OrthonormalBasis::
115        ::CosTheta(wh), 3.0f));
115    float ab = powf(OrthonormalBasis::CosPhi(wh), 2) / (nx * nx);
116    float ac = powf(OrthonormalBasis::SinPhi(wh), 2) / (ny * ny);
117
118    float b = -tanTheta2 * (ab + ac);
119    return a * expf(b);
120 }

```

2.4 Camera

2.4.1 ApertureShape

Listing 2.35: ApertureShape header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Math/Statistics.h"
4 #include "../Textures/BitmapTexture.h"
5
6 class ApertureShape
7 {
8 public:
9     ApertureShape(Bitmap apertureShape)
10        : texture(new BitmapTexture(apertureShape)),
11          distribution(&apertureShape)
12    {
13    }
14
15    Point2D Sample(Point2D u) const;
16
17 protected:
18     BitmapTexture* texture;
19     Distribution2D distribution;
20 };
21
22 inline Point2D ApertureShape::Sample(Point2D u) const
23 {
24     Point2D sample;
25     float pdf;
26     distribution.Sample2D(u, sample, pdf);
27     return Point2D(sample[0], 1.0f - sample[1]);
28 }
```

2.4.2 OrthographicCamera

Listing 2.36: OrthographicCamera header file

```
1 #pragma once
2 #include "../Camera.h"
3
4 class OrthographicCamera : public Camera
5 {
6 public:
7     OrthographicCamera(float fov, float aspectRatio, float
8         targetDistance, float zNear, float zFar);
9     Ray CalculateRay(float x, float y, float lensX, float lensY, int
10        width, int height) const override;
11     virtual void UpdateCamera() override;
12 protected:
13     float AspectRatio;
14     float FOV;
15     float TargetDistance;
16 }
```

Listing 2.37: OrthographicCamera source file

```
1 #include "OrthographicCamera.h"
2
3
4 OrthographicCamera::OrthographicCamera(float fov, float aspectRatio,
5   float targetDistance, float zNear, float zFar)
6   :FOV{ fov }, AspectRatio{ aspectRatio }, TargetDistance{
7     targetDistance}
8 {
9   CameraToWorld = Matrix::Orthographic(fov, aspectRatio,
10    targetDistance, zNear, zFar);
11 }
12 Ray OrthographicCamera::CalculateRay(float x, float y, float lensX,
13   float lensY, int width, int height) const
14 {
15   float NDCX = x / width;
16   float NDCY = y / height;
17   // NDC -> Screen transformation
18   float ScreenSpaceX = (2 * NDCX - 1);
19   float ScreenSpaceY = (2 * NDCY - 1);
20   Point u = Point(ScreenSpaceX, -ScreenSpaceY, -1);
21   Vector v(u.GetX(), u.GetY(), u.GetZ());
22   Ray r(Point(0, 0, TargetDistance), v);
23   return CameraToWorld * r;
24 }
25 void OrthographicCamera::UpdateCamera()
26 {
27 }
```

2.4.3 PinholeCamera

Listing 2.38: PinholeCamera header file

```
1 #pragma once
2 #include "../Camera.h"
3
4 class PinholeCamera : public Camera
5 {
6 public:
7     PinholeCamera();
8     PinholeCamera(Point eye, Point target, Vector up, float fov, float
9         aspectRatio, float zNear, float zFar);
10    PinholeCamera(Matrix& cameraToWorld, float fov, float aspectRatio,
11        float zNear, float zFar);
12    virtual Ray CalculateRay(float x, float y, float lensX, float lensY
13        , int width, int height) const override;
14    virtual void UpdateCamera() override;
15
16 private:
17     float AspectRatio;
18     float FOV;
19     Matrix Projection;           // The camera->screen transform
20     Matrix InvProjection;       // The screen->camera transform
21 };
```

Listing 2.39: PinholeCamera source file

```
1 #include "PinholeCamera.h"
2 #include "../../Core/Utility/Misc.h"
3 #include "../../Core/Math/Matrix.h"
4
5 PinholeCamera::PinholeCamera()
6     : Camera{ Point(0, 0, 0), Point(0, 0, -1), Vector(0, 1, 0) }, FOV
7         (45.0), AspectRatio(1)
8 {}
9
10 PinholeCamera::PinholeCamera(Point eye, Point target, Vector up, float
11     fov, float aspectRatio, float zNear, float zFar)
12     : Camera{ eye, target, up }, FOV{ fov }, AspectRatio{aspectRatio}
13 {
14     CameraToWorld = Matrix::LookAt(eye, target, up);
15     CameraToWorld = Matrix::Inverse(CameraToWorld);
16     Projection = Matrix::Perspective(fov, aspectRatio, zNear, zFar);
17     InvProjection = Matrix::Inverse(Projection);
18
19     std::cout << "Projection" << std::endl;
20     std::cout << Projection << std::endl;
21     std::cout << "Inverse" << std::endl;
22     std::cout << InvProjection << std::endl;
23 }
24
25 PinholeCamera::PinholeCamera(Matrix& cameraToWorld, float fov, float
26     aspectRatio, float zNear, float zFar)
27     : Camera(cameraToWorld), FOV(fov), AspectRatio(aspectRatio)
28 {
29     Projection = Matrix::Perspective(fov, aspectRatio, zNear, zFar);
30     InvProjection = Matrix::Inverse(Projection);
31 }
32
33 Ray PinholeCamera::CalculateRay(float x, float y, float lensX, float
34     lensY, int width, int height) const
35 {
36     float NDCX = x / width;
37     float NDCY = y / height;
38     // NDC -> Screen transformation
39     float ScreenSpaceX = (2 * NDCX - 1);
40     float ScreenSpaceY = (2 * NDCY - 1);
41     Point u = InvProjection * Point(ScreenSpaceX, -ScreenSpaceY, 0);
42     return CameraToWorld * Ray(Point(0, 0, 0), Vector(u).Hat());
43 }
44
45 void PinholeCamera::UpdateCamera()
46 {
```

2.4.4 ThinLensCamera

Listing 2.40: ThinLensCamera header file

```
1 #pragma once
2 #include "../Camera.h"
3 #include "../../Core/Math/Point2D.h"
4 #include "../Sample.h"
5 #include "ApertureShape.h"
6
7 class ThinLensCamera : public Camera
8 {
9 public:
10     ThinLensCamera();
11     ThinLensCamera(Point eye, Point target, Vector up, float fov, float
12                     aspectRatio, float focalLength, float lensRadius);
12     ThinLensCamera(Matrix& cameraToWorld, float fov, float aspectRatio,
13                     float focalDistance, float lensRadius, ApertureShape* shape =
14                     nullptr);
13     virtual Ray CalculateRay(float x, float y, float lensX, float lensY
14                               , int width, int height) const override;
14
15 private:
16     float AspectRatio;
17     float FOV;
18     float FocalDistance;
19     float LensRadius;
20     Matrix Projection;           // The camera->screen transform
21     Matrix InvProjection;        // The screen->camera transform
22     Random::RandomDouble rng;
23     ApertureShape* Shape;
24 };
25
26 inline ThinLensCamera::ThinLensCamera()
27     : Camera{ Point(0, 0, 0), Point(0, 0, -1), Vector(0, 1, 0) }, FOV
28         (45.0), AspectRatio(1.33f), FocalDistance(1.0f), LensRadius
29         (0.001f), Shape(nullptr)
30 {
31     inline ThinLensCamera::ThinLensCamera(Point eye, Point target, Vector
32                                         up, float fov, float aspectRatio, float
33                                         focalDistance, float
34                                         lensRadius)
35         : FocalDistance(focalDistance), LensRadius(lensRadius), AspectRatio
36             (aspectRatio), FOV(fov), Shape(nullptr)
37 {
38     CameraToWorld = Matrix::LookAt(eye, target, up);
39     CameraToWorld = Matrix::Inverse(CameraToWorld);
40     // Camera -> Screen
41     Projection = Matrix::Perspective(fov, aspectRatio, 0.1f, 1000.0f);
42     // Screen -> Camera
43     InvProjection = Matrix::Inverse(Projection);
44 }
```

```

41
42 inline ThinLensCamera::ThinLensCamera(Matrix& cameraToWorld, float fov,
43     float aspectRatio, float focalDistance, float lensRadius,
44     ApertureShape* shape)
45     :FocalDistance(focalDistance), LensRadius(lensRadius), AspectRatio(
46         aspectRatio), FOV(fov), Shape(shape)
47
48 }
49
50 inline Ray ThinLensCamera::CalculateRay(float x, float y, float lensX,
51     float lensY, int width, int height) const
52 {
53     float NDCX = x / width;
54     float NDCY = y / height;
55     // NDC -> Screen transformation
56     float ScreenSpaceX = (2 * NDCX - 1);
57     float ScreenSpaceY = (2 * NDCY - 1);
58
59     // Screen-> Camera transformation
60     Point CameraSample = InvProjection * Point(ScreenSpaceX, -
61         ScreenSpaceY, 0);
62     Ray CameraLocalRay(Point(0, 0, 0), Vector(CameraSample));
63
64     // Generate sample on the lens
65     Point2D LensSample(rng(), rng());
66     if (Shape != nullptr)
67     {
68         LensSample = Shape->Sample(LensSample) * LensRadius;
69     }
70     else
71     {
72         LensSample = Sample::DiscConcentricSample(LensSample) *
73             LensRadius;
74     }
75
76     // Find the intersection with focal plane
77     float t = -FocalDistance / CameraLocalRay.GetDirection().GetZ();
78     Point focalPoint = CameraLocalRay(t);
79     Point LensPoint(LensSample.GetX(), LensSample.GetY(), 0.0f);
80     Ray lensRay(LensPoint, Vector(focalPoint - LensPoint).Hat());
81     return CameraToWorld * lensRay;
82 }
```

2.5 Geometry

2.5.1 Box

Listing 2.41: Box header file

```
1 #pragma once
2 #include "../Geometry.h"
3
4 class Box : public Geometry
5 {
6 public:
7     Box(Point& LP, Point& RP);
8     virtual bool IntersectRay(const Ray& ray, float& tMin, float&
9         epsilon, Intersection& intersection) const override;
10 private:
11     // Diagonal points representing the box
12     Point P0;
13     Point P1;
14 };
```

Listing 2.42: Box source file

```
1 #include "Box.h"
2 #include "../Ray.h"
3 #include "../Intersection.h"
4 #include "../../Core/Math/Point.h"
5 #
6 Box::Box(Point& LP, Point& RP)
7     :P0{ LP }, P1{RP}
8 {}
9
10 // See http://www.cs.utah.edu/~awilliam/box/box.pdf
11 // Uses slabs (parallel planes) to determine intersection by clipping
12 // each dimension and
13 // finding the overlapping intervals between each pair of slab.
14 bool Box::IntersectRay(const Ray& ray, float& tMin, float& epsilon,
15                         Intersection& intersection) const
16 {
17     float TMin, TMax;
18     float TYMin, TYMax;
19     float TZMin, TZMax;
20
21     Vector Direction = ray.GetDirection();
22     Point Origin = ray.GetOrigin();
23
24     if (Direction.GetX() >= 0)
25     {
26         TMin = (P0.GetX() - Origin.GetX()) / Direction.GetX();
27         TMax = (P1.GetX() - Origin.GetX()) / Direction.GetX();
28     }
29     else
30     {
31         TMin = (P1.GetX() - Origin.GetX()) / Direction.GetX();
32         TMax = (P0.GetX() - Origin.GetX()) / Direction.GetX();
33     }
34
35     if (Direction.GetY() >= 0)
36     {
37         TYMin = (P0.GetY() - Origin.GetY()) / Direction.GetY();
38         TYMax = (P1.GetY() - Origin.GetY()) / Direction.GetY();
39     }
40     else
41     {
42         TYMin = (P1.GetY() - Origin.GetY()) / Direction.GetY();
43         TYMax = (P0.GetY() - Origin.GetY()) / Direction.GetY();
44     }
45     if ((TMin > TYMax) || (TYMin > TMax))
46         return false;
47
48     if (TYMin > TMin)
49         TMin = TYMin;
```

```
50         TMax = TYMax;
51
52     if (Direction.GetZ() >= 0)
53     {
54         TZMin = (P0.GetZ() - Origin.GetZ()) / Direction.GetZ();
55         TZMax = (P1.GetZ() - Origin.GetZ()) / Direction.GetZ();
56     }
57     else
58     {
59         TZMin = (P1.GetZ() - Origin.GetZ()) / Direction.GetZ();
60         TZMax = (P0.GetZ() - Origin.GetZ()) / Direction.GetZ();
61     }
62     if ((TMin > TZMax) || (TZMin > TMax))
63         return false;
64
65     if (TZMin > TMin)
66         TMin = TZMin;
67
68     if (TZMax < TMax)
69         TMax = TZMax;
70
71     return (TMin > epsilon);
72
73 }
```

2.5.2 Disc

Listing 2.43: Disc header file

```
1 #pragma once
2
3 #include "../Sample.h"
4 #include "../Geometry.h"
5 #include "../../Core/Math/Transform.h"
6 #include "../../Core/Math/Point2D.h"
7 class Disc : public Geometry
8 {
9 public:
10
11     Disc(Transform* objToWorld, Transform* worldToObject, float yHeight
12         , float radius, float phiMax);
13     ~Disc() override;
14
15     Point Sample(const Point2D& sample, Normal& normal) const override;
16
17     float SurfaceArea() const override;
18     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
19                         Intersection& intersection) const override;
20
21     BBox GetBounds() const override;
22
23 private:
24     float Radius;
25     float Height;
26     float PhiMax;
27 };
28
29
30 }
31
32 inline BBox Disc::GetBounds() const
33 {
34     return *ObjectToWorld * BBox(Point(-Radius, Height, -Radius), Point
35         (Radius, Height, Radius));
36 }
37
38 inline Disc::~Disc()
39 {
40
41     inline Point Disc::Sample(const Point2D& sample, Normal& normal) const
42     {
43         Point2D point(Sample::DiscConcentricSample(sample));
44         normal = (*ObjectToWorld * Normal(0, -1, 0)).Hat();
```

```

45     return *ObjectToWorld * Point(point[0] * Radius, 0, point[1] *
        Radius);
46 }
47
48 inline float Disc::SurfaceArea() const
49 {
50     return (DegToRad(PhiMax) * 0.5 * (Radius * Radius));
51 }
52
53 inline bool Disc::IntersectRay(const Ray& ray, float& tMin, float&
    epsilon, Intersection& intersection) const
54 {
55     // Transform ray into object space
56     Ray localRay = *WorldToObject * ray;
57
58     // Ray is parallel to disc
59     if (fabsf(localRay.GetDirection().GetY()) < std::numeric_limits<
        float>::epsilon())
60     {
61         return false;
62     }
63
64     float t = (Height - localRay.GetOrigin().GetY()) / localRay.
        GetDirection().GetY();
65     if (t < epsilon)
66     {
67         return false;
68     }
69
70     Point HitPoint = localRay(t);
71     float Distance = HitPoint.GetX() * HitPoint.GetX() + HitPoint.GetZ
        () * HitPoint.GetZ();
72     if (Distance > Radius * Radius)
73     {
74         return false;
75     }
76
77     float phi = atan2f(HitPoint.GetZ(), HitPoint.GetX());
78     if (phi < 0)
79     {
80         phi += 2.0 * M_PI;
81     }
82     if (phi > DegToRad(PhiMax))
83     {
84         return false;
85     }
86     if (t < tMin)
87     {
88         tMin = t;
89         float u = phi / DegToRad(PhiMax);
90         float v = 1.0f - (sqrtf(Distance) / Radius);
91

```

```
92     //Point p = *ObjectToWorld * HitPoint;
93     Point p = ray(t);
94     intersection.IGeometry = dynamic_cast<Geometry*>(const_cast<
95         Disc*>(this));
96     intersection.IPoint = p;
97     intersection.INormal = (*ObjectToWorld * Normal(0, -1, 0)).Hat
98         ();
99     intersection.ShadingBasis = OrthonormalBasis::FromW(Vector(
100         intersection.INormal));
101    intersection.U = u;
102    intersection.V = v;
103    intersection.U0 = u;
104    intersection.V0 = v;
105    return true;
106 }
```

2.5.3 Plane

Listing 2.44: Plane header file

```
1 #pragma once
2
3 #include "../Geometry.h"
4 #include "../../Core/Math/Point.h"
5 #include "../../Core/Math/Vector.h"
6
7 class Plane : public Geometry
8 {
9
10 public:
11     Plane(Point& Position, Normal Normal)
12         :Position{ Position }, PNormal{ Normal }){}
13     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
14                       Intersection& intersection) const override;
15
16 private:
17     Normal PNormal;
18     Point Position;
19 }
```

Listing 2.45: Plane source file

```
1 #include "Plane.h"
2
3 // Equation of Plane in Point-Normal Form is n.(p-p_0)=0 where p_0 is
4 // some point on the plane
5 // and n is normal to the plane, then p is all the points on the plane.
6 bool Plane::IntersectRay(const Ray& ray, float& tMin, float& epsilon,
7     Intersection& intersection) const
8 {
9     float T;
10    float Numerator = Dot(PNormal, Position - ray.GetOrigin());
11    float Denominator = Dot(PNormal, ray.GetDirection());
12    T = (Numerator / Denominator);
13    if (T > epsilon)
14    {
15        intersection.SetNormal(PNormal);
16        intersection.SetPoint(ray.GetOrigin() + T * ray.GetDirection())
17        ;
18    }
19    return false;
}
```

2.5.4 Quadrilateral

Listing 2.46: Quadrilateral header file

```
1 #pragma once
2
3 #include "../Geometry.h"
4 #include "../../Core/Math/Transform.h"
5 #include "../../Core/Math/Point2D.h"
6
7 // -----
8 // |           |
9 // |           | Width
10 // |-----|
11 // Length
12 // z |-- x
13 // Default position points downwards
14 // -----
15 //       |
16 //       v
17 class Quadrilateral : public Geometry
18 {
19 public:
20     Quadrilateral(Transform* objToWorld, Transform* worldToObject,
21                    float length, float width);
22     ~Quadrilateral() override;
23
24     float Pdf(const Point& sample) const override;
25     Point Sample(const Point2D& sample, Normal& normal) const override;
26     float SurfaceArea() const override;
27
28     BBox GetBounds() const override;
29     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
30                        Intersection& intersection) const override;
31
32 private:
33     // TODO: This is actually half width/height.
34     float Width;
35     float Length;
36 };
37
38 inline Quadrilateral::Quadrilateral(Transform* objToWorld, Transform*
39                                     worldToObject, float length, float width)
40 : Geometry(objToWorld, worldToObject),
41            Length(length),
42            Width(width)
43 {
44     inline Quadrilateral::~Quadrilateral()
45 {
46 }
```

```

47 }
48
49 inline float Quadrilateral::Pdf(const Point& sample) const
50 {
51     return 1 / SurfaceArea();
52 }
53
54 inline Point Quadrilateral::Sample(const Point2D& sample, Normal&
55     normal) const
56 {
57     normal = (*ObjectToWorld * Normal(0, -1, 0)).Hat();
58     return *ObjectToWorld * Point((2.0f *sample.GetX() - 1.0f) * Length
59         , 0, (2.0f * sample.GetY() - 1.0f) * Width);
60 }
61
62 inline float Quadrilateral::SurfaceArea() const
63 {
64     return ((Width + Width) * (Length + Length));
65 }
66
67 inline BBox Quadrilateral::GetBounds() const
68 {
69     return *ObjectToWorld * BBox(Point(-Length, 0, -Width), Point(
70         Length, 0, Width));
71 }
72
73 inline bool Quadrilateral::IntersectRay(const Ray& ray, float& tMin,
74     float& epsilon, Intersection& intersection) const
75 {
76     // Transform ray into object space
77     Ray localRay = *WorldToObject * ray;
78     float Height = 0.0f;
79     if (fabsf(localRay.GetDirection().GetY()) < std::numeric_limits<
80         float>::epsilon())
81     {
82         return false;
83     }
84
85     float t = (Height - localRay.GetOrigin().GetY()) / localRay.
86     GetDirection().GetY();
87     if (t < epsilon)
88     {
89         return false;
90     }
91
92     HitPoint = localRay(t);
93     float DistanceX = fabsf(HitPoint.GetX());
94     float DistanceZ = fabsf(HitPoint.GetZ());
95     if (DistanceX > Length || DistanceZ > Width)
96     {
97         return false;
98     }

```

```

93
94     if (t < tMin)
95     {
96         tMin = t;
97
98         float u = (HitPoint.GetX()/Length + 1) / 2;
99         float v = 1 - (HitPoint.GetZ()/Width + 1) / 2;
100        //Point p = *ObjectToWorld * HitPoint;
101        Point p = ray(t);
102        intersection.IGeometry = (Geometry*)this;
103        intersection.IPoint = p;
104        intersection.INormal = (*ObjectToWorld * Normal(0, -1, 0)).Hat
105            ();
106        intersection.ShadingBasis = OrthonormalBasis::FromW(Vector(
107            intersection.INormal));
108        intersection.U = u;
109        intersection.V = v;
110        intersection.U0 = u;
111        intersection.V0 = v;
112        return true;
113    }
114    return false;
115 }
```

2.5.5 Sphere

Listing 2.47: Sphere header file

```
1 #pragma once
2
3 #include "../Geometry.h"
4 #include "../../Core/Math/Point.h"
5 #include "../../Core/Math/Vector.h"
6 #include "../../Ray.h"
7 #include "../../Intersection.h"
8 class Sphere : public Geometry
9 {
10 public:
11     Sphere(Transform* objToWorld, Transform* worldToObject, float
12             Radius);
13     ~Sphere() override;
14
15     float SurfaceArea() const override;
16     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
17                       Intersection& intersection) const override;
18
19     Point Sample(const Point2D& sample, Normal& normal) const override;
20
21     float GetRadius() {return Radius;}
22     Point GetPosition() { return Position; }
23     virtual BBox GetBounds() const override;
24
25 private:
26     float Radius;
27     Point Position;
28     BBox Bounds;
29 };
```

2.5.6 Triangle

Listing 2.48: Triangle header file

```
1 #pragma once
2
3 #include "../Geometry.h"
4 #include "../../Core/Math/Point.h"
5 #include "../../Core/Math/Vector.h"
6
7 #include "TriangleMesh.h"
8
9 class Triangle : public Geometry
10 {
11 public:
12
13     Triangle(Transform* objectToWorld, Transform* worldToObject,
14             TriangleMesh* mesh, Point& P0, Point& P1, Point& P2)
15         : Geometry(objectToWorld, worldToObject), ParentMesh(mesh), P0{
16             P0 }, P1{ P1 }, P2{ P2 }, TNormal{1,0,0}
17     {
18         Bounds = Bounds + P0;
19         Bounds = Bounds + P1;
20         Bounds = Bounds + P2;
21     }
22     ~Triangle() override;
23
24     Point Sample(const Point2D& sample, Normal& normal) const override;
25     float SurfaceArea() const override;
26     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
27                       Intersection& intersection) const override;
28
29     virtual BBox GetBounds() const override;
30     Normal GetNormal() const { return TNormal; }
31     Point U0, U1, U2;
32     Normal N0, N1, N2;
33
34     private:
35     Point P0, P1, P2;
36     Normal TNormal;
37     TriangleMesh* ParentMesh;
38
39     BBox Bounds;
40 };
```

Listing 2.49: Triangle source file

```
1 #include "Triangle.h"
2 #include "../Sample.h"
3
4 BBox Triangle::GetBounds() const
5 {
6     return Bounds;
7 }
8
9 Triangle::~Triangle()
10 {
11 }
12
13 Point Triangle::Sample(const Point2D& sample, Normal& normal) const
14 {
15     Point2D uniformPoint(Sample::UniformTriangle(sample));
16     Point p = (P2 * uniformPoint[0]) + (P1 * uniformPoint[1]) + (P0 *
17         (1 - uniformPoint[0] - uniformPoint[1]));
18     normal = Normal(Cross(P1 - P0, P2 - P0).Hat());
19     return p;
20 }
21
22 float Triangle::SurfaceArea() const
23 {
24     return 0.5f * Cross(P1 - P0, P2 - P0).Length();
25 }
26
27 //Moller-Trumbore Triangle-Ray intersection
28 //http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/
29 //Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf
30 bool Triangle::IntersectRay(const Ray& ray, float& tMin, float& epsilon
31 , Intersection& intersection) const
32 {
33     double T;
34     Vector Edge1 = P1 - P0;
35     Vector Edge2 = P2 - P0;
36
37     Vector PVec = Cross(ray.GetDirection(), Edge2);
38
39     double Determinant = Dot(Edge1, PVec);
40     #if 0
41         // WITH CULLING
42         if (Determinant < epsilon)
43             return false;
44
45         // Distance from P0 to ray Origin
46         Vector TVec = ray.GetOrigin() - P0;
47
48         double U = Dot(TVec, PVec);
49         if (U < 0.0 || U > Determinant)
50             return false;
```

```

49
50     Vector QVec = Cross(TVec, Edge1);
51     double V = Dot(ray.GetDirection(), QVec);
52     if (V < 0.0 || U + V > Determinant)
53         return false;
54
55     T = Dot(Edge2, QVec);
56     double InverseDeterminant = 1.0 / Determinant;
57     T *= InverseDeterminant;
58     U *= InverseDeterminant;
59     V *= InverseDeterminant;
60     // END CULLING
61 #else
62     if (Determinant > -epsilon && Determinant < epsilon)
63     {
64         return false;
65     }
66     double InverseDeterminant = 1.0 / Determinant;
67
68     Vector TVec = ray.GetOrigin() - P0;
69     float U = Dot(TVec, PVec) * InverseDeterminant;
70     if (U < 0.0 || U > 1.0)
71     {
72         return false;
73     }
74
75     Vector QVec = Cross(TVec, Edge1);
76
77     double V = Dot(ray.GetDirection(), QVec) * InverseDeterminant;
78     if (V < 0.0 || U + V > 1.0)
79     {
80         return false;
81     }
82
83     T = Dot(Edge2, QVec) * InverseDeterminant;
84
85 #endif
86     Point uvw = U0 * (1 - U - V) + U1 * U + U2 * V;
87     if (ParentMesh->HasAlphaMap() &&
88         ParentMesh->GetAlphaMap()->GetValue(uvw[0], uvw[1],
89             intersection) == Colour(0.0f, 0.0f, 0.0f))
90     {
91         return false;
92     }
93
94     if (T < ray.MinT || T > ray.MaxT)
95     {
96         return false;
97     }
98     if (T > 1e-2 && T < tMin)
99     {
100         tMin = T;

```

```

100     Normal normal = ((1 - U - V) * NO) + (U * N1) + (V*N2);
101     // signed normals
102     intersection.SetNormal(normal);
103     // unsigned normals
104     //intersection.SetNormal(Normal(normal.GetX() * 0.5 + 0.5,
105     //                                normal.GetY()* 0.5 + 0.5, normal.GetZ()* 0.5 + 0.5));
106     // abs normals
107     //intersection.SetNormal(Normal(fabs(normal.GetX()), fabs(
108     //                                normal.GetY()), fabs(normal.GetZ())));
109     intersection.IPoint = ray(T);
110     intersection.U = uvw.GetX();
111     intersection.V = uvw.GetY();
112     intersection.U0 = U;
113     intersection.V0 = V;
114     intersection.N = Normal(Cross(Edge1, Edge2).Hat());
115     intersection.IGeometry = (Geometry*)this;
116     intersection.ShadingBasis = OrthonormalBasis::FromW(Vector(
117     normal));
118     return true;
119 }
120 return false;
121 }
```

2.5.7 TriangleMesh

Listing 2.50: TriangleMesh header file

```
1 #pragma once
2
3 #include "../Geometry.h"
4 #include "../Texture.h"
5 #include "../BBox.h"
6 #include "../../Core/Math/Point.h"
7 #include <vector>
8 #include "../../Core/Math/Statistics.h"
9
10 //struct Vertex
11 //{
12 //    Point VVertex;
13 //    Normal VNormal;
14 //};
15 struct Vertex
16 {
17     Vertex(int pIndex, int nIndex, int uvwIndex)
18         : PositionIndex(pIndex), NormalIndex(nIndex), UVWIndex(uvwIndex)
19     {}
20     int PositionIndex;
21     int NormalIndex;
22     int UVWIndex;
23 };
24
25 struct VertexInformation
26 {
27     Point position;
28     Normal normal;
29     Point uvw;
30 };
31
32 class Texture;
33
34 class TriangleMesh : public Geometry
35 {
36 public:
37     //TriangleMesh(Transform* objectToWorld, Transform* worldToObject,
38     //              std::vector<int>& indices, std::vector<Point>& points, std::vector<Normal>& normals, std::vector<Point>& uvws, Texture*
39     //              alphaTexture = nullptr);
40     TriangleMesh(Transform* objectToWorld, Transform* worldToObject,
41                  std::vector<Vertex>& indices, std::vector<Point>& points, std::vector<Normal>& normals, std::vector<Point>& uvws, Texture*
42                  alphaTexture = nullptr);
43     TriangleMesh(Transform* objectToWorld, Transform* worldToObject,
44                  std::vector<VertexInformation>& vertices, Texture* alphaTexture
45                  = nullptr);
46 }
```

```

41     TriangleMesh(const TriangleMesh& TMesh);
42
43     float SurfaceArea() const override;
44     virtual bool TriangleMesh::IntersectRay(const Ray& ray, float& tMin
45         , float& epsilon, Intersection& intersection) const override;
46
47     std::vector<Point> GetPoints() const;
48     std::vector<Geometry*> GetTriangles() const;
49     void SetUVW(std::vector<Point>& p) { UVWs = p; }
50     std::vector<Point> GetUVWs() const {return UVWs;}
51
52     virtual Geometry* GetSubGeometry(int i) override;
53     virtual int GetNumberOfGeometries() override;
54     virtual BBox GetBounds() const override;
55     const Texture* GetAlphaMap() const { return AlphaMap; }
56
57     bool IsCompound() const override;
58     bool HasAlphaMap() const { return AlphaMap != nullptr; }
59
60     Point Sample(const Point2D& sample, Normal& normal) const override;
61     void CalculateAreaDistribution();
62
63 private:
64     int TriangleCount;
65     int VertexCount;
66     std::vector<Vertex> Indices;
67     std::vector<Point> Points;
68     std::vector<Point> UVWs;
69     std::vector<Normal> Normals;
70     std::vector<Geometry*> WorldTriangles;
71     Texture* AlphaMap;
72     std::vector<Point> WorldPoints;
73     Distribution1D* AreaDistribution;
74     float Area;
75 };
76 inline std::vector<Point> TriangleMesh::GetPoints() const { return
77     Points; }
78 inline std::vector<Geometry*> TriangleMesh::GetTriangles() const {
    return WorldTriangles; }

```

Listing 2.51: TriangleMesh source file

```

1 #include "TriangleMesh.h"
2
3 #include "../Geometry/Triangle.h"
4 #include "../Sample.h"
5
6 TriangleMesh::TriangleMesh(Transform* objectToWorld, Transform*
7     worldToObject, std::vector<Vertex>& indices, std::vector<Point>&
8     points, std::vector<Normal>& normals, std::vector<Point>& uvws,
9     Texture* alphaTexture)
10    : Geometry(objectToWorld, worldToObject), Indices(indices), Points(
11        points), Normals(normals), UVWs(uvws), AlphaMap(alphaTexture)
12 {
13     //material = m;
14     WorldPoints.reserve(points.size());
15     for (int i = 0; i < points.size(); ++i)
16     {
17         Point WorldPoint = *ObjectToWorld * Points[i];
18         WorldPoints.push_back(WorldPoint);
19     }
20
21     WorldTriangles.reserve(indices.size());
22     TriangleCount = 0;
23     for (int i = 0; i < indices.size(); i += 3)
24     {
25         Vertex v1 = indices[i];
26         Vertex v2 = indices[i + 1];
27         Vertex v3 = indices[i + 2];
28         Triangle* triangle = new Triangle(objectToWorld, worldToObject,
29             reinterpret_cast<TriangleMesh*>(const_cast<TriangleMesh*>(
30                 this)), WorldPoints[v1.PositionIndex], WorldPoints[v2.
31                 PositionIndex], WorldPoints[v3.PositionIndex]);
32         BoundingBox += triangle->GetBounds();
33         triangle->U0 = uvws[v1.UVWIndex];
34         triangle->U1 = uvws[v2.UVWIndex];
35         triangle->U2 = uvws[v3.UVWIndex];
36         triangle->N0 = normals[v1.NormalIndex];
37         triangle->N1 = normals[v2.NormalIndex];
38         triangle->N2 = normals[v3.NormalIndex];
39         //triangle->material = m;
40         WorldTriangles.push_back(triangle);
41         ++TriangleCount;
42     }
43     CalculateAreaDistribution();
44 }
45
46 TriangleMesh::TriangleMesh(Transform* objectToWorld, Transform*
47     worldToObject, std::vector<VertexInformation>& vertices, Texture*
48     alphaTexture)
49    : Geometry(objectToWorld, worldToObject), AlphaMap(alphaTexture)
50 {
51     for (const VertexInformation& vert : vertices)

```

```

43     {
44         BoundingBox = BoundingBox + vert.position;
45     }
46     for (int i = 0; i < vertices.size(); ++i)
47     {
48         Point WorldPoint = *ObjectToWorld * vertices[i].position;
49         WorldPoints.push_back(WorldPoint);
50     }
51     TriangleCount = 0;
52     for (int i = 0; i < vertices.size(); i += 3)
53     {
54         Triangle* triangle = new Triangle(objectToWorld, worldToObject,
55                                         reinterpret_cast<TriangleMesh*>(const_cast<TriangleMesh*>(
56                                         this)), WorldPoints[i], WorldPoints[i+1], WorldPoints[i+2])
57         ;
58         BoundingBox += triangle->GetBounds();
59         triangle->U0 = vertices[i].uvw;
60         triangle->U1 = vertices[i + 1].uvw;
61         triangle->U2 = vertices[i + 2].uvw;
62         triangle->N0 = vertices[i].normal;
63         triangle->N1 = vertices[i + 1].normal;
64         triangle->N2 = vertices[i + 2].normal;
65         WorldTriangles.push_back(triangle);
66         ++TriangleCount;
67     }
68     CalculateAreaDistribution();
69 }
70 TriangleMesh::TriangleMesh(const TriangleMesh& m)
71 : Geometry(m.ObjectToWorld, m.WorldToObject), Indices(m.Indices),
72   Points(m.Points), Normals(m.Normals), UVWs(m.UVWs), AlphaMap(m.
73   AlphaMap)
74 {
75     //material = m.material;
76 }
77 float TriangleMesh::SurfaceArea() const
78 {
79     return Area;
80 }
81 bool TriangleMesh::IntersectRay(const Ray& ray, float& tMin, float&
82                                 epsilon, Intersection& intersection) const
83 {
84     for (int i = 0; i < WorldTriangles.size(); ++i)
85     {
86         if (WorldTriangles[i]->IntersectRay(ray, tMin, epsilon,
87                                             intersection))
88         {
89             return true;
90         }

```

```

88     }
89     return false;
90 }
91
92
93 Geometry* TriangleMesh::GetSubGeometry(int i)
94 {
95     return WorldTriangles[i];
96 }
97
98 int TriangleMesh::GetNumberOfGeometries()
99 {
100    return TriangleCount;
101 }
102
103 BBox TriangleMesh::GetBounds() const
104 {
105    return BoundingBox;
106 }
107
108 bool TriangleMesh::IsCompound() const
109 {
110    return true;
111 }
112
113 Point TriangleMesh::Sample(const Point2D& sample, Normal& normal) const
114 {
115     float x;
116     float pdf;
117     AreaDistribution->Sample1D(sample[0], x, pdf);
118     int index = clamp(x * WorldTriangles.size(), 0.0f, float(
119         WorldTriangles.size() - 1));
120     Triangle* triangle = (Triangle*)WorldTriangles[index];
121     Point p = (triangle->Sample(sample, normal));
122     return p;
123 }
124
125 void TriangleMesh::CalculateAreaDistribution()
126 {
127     Area = 0;
128     std::vector<float> areas;
129     for (int i = 0; i < WorldTriangles.size(); ++i)
130     {
131         float temp = WorldTriangles[i]->SurfaceArea();
132         Area += temp;
133         areas.push_back(temp);
134     }
135     AreaDistribution = new Distribution1D(areas);
}

```

2.6 Integrators

2.6.1 AVIntegrator

Listing 2.52: AVIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 //#include "../Materials/DiffuseMaterial.h"
5 //#include "../Materials/PhongMaterial.h"
6 #include "../Material.h"
7 #include "../Light.h"
8 #include "../Scene.h"
9 #include "../Sample.h"
10
11 class AVIntegrator : public Integrator
12 {
13 public:
14     AVIntegrator(float length)
15         : RayLength(length)
16     {}
17
18
19     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler* s)
20         const override
21     {
22         Intersection intersection;
23         if (scene->Intersect(ray, intersection))
24         {
25             Point hitpoint = intersection.GetPoint();
26             Normal N = intersection.GetNormal();
27             Random::RandomDouble rng(0, 1, rand());
28             Vector sample = Sample::UniformHemisphereSample(rng, N);
29             Ray r(hitpoint, sample);
30             if (scene->Intersect(r, intersection))
31             {
32                 return Colour(0.0f, 0.0f, 0.0f);
33             }
34         }
35         return Colour(1.0f, 1.0f, 1.0f);
36     }
37
38
39
40 private:
41     float RayLength;
42 };
```

2.6.2 DirectIntegrator

Listing 2.53: DirectIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Light.h"
6 #include "../Scene.h"
7 #include "../Sample.h"
8
9 class DirectIntegrator : public Integrator
10 {
11 public:
12     DirectIntegrator()
13     {}
14
15
16     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler* s)
17         const override
18     {
19         Intersection intersection;
20         if (scene->Intersect(ray, intersection))
21         {
22             Colour color(0.0f);
23             Point point = intersection.GetPoint();
24             Normal normal = Normal(intersection.ShadingBasis.W);
25             const Material* bsdf = intersection.material;
26             Normal n;
27             Vector wo = -ray.GetDirection();
28             for (Light* emitter : scene->GetLights())
29             {
30                 Vector wi;
31                 Point lightSample;
32                 Colour power = emitter->SampleLight(s->Sample2D(),
33                     lightSample, wi, intersection);
34                 Ray r(point, Normalize(wi));
35                 if (scene->Intersect(r, intersection) &&
36                     DistanceSquared(intersection.GetPoint(),
37                         intersection.GetPoint()) + 1e-6 < wi.LengthSquared()
38                     )
39                 {
40                     continue;
41                 }
42                 float pdf;
43                 color += power * bsdf->SampleF(intersection, wo, wi,
44                     pdf) * std::max(0.0f, Dot(wi, normal));
45             }
46             return color;
47         }
48         Colour Le;
```

```
44
45     for (Light* emitter : scene->GetLights())
46     {
47         if (emitter->IsInfinite())
48         {
49             Le += emitter->Le(ray, intersection);
50         }
51     }
52     return Le;
53 }
54
55 };
```

2.6.3 DirectLightingIntegrator

Listing 2.54: DirectLightingIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Scene.h"
6
7 class DirectLightingIntegrator : public Integrator
8 {
9 public:
10     DirectLightingIntegrator()
11     {
12     }
13
14
15     bool IsInShadow(const Scene* scene, const Ray& ray, Light* light,
16                      Point& lightPos) const
17     {
18         Intersection is;
19         Intersection iss;
20         float tCurr = INFINITY;
21         float tMin = INFINITY;
22         bool hit = false;
23         // Need to find a good value otherwise you get speckled shadows
24         float epsilon = 1e-6;
25         if (scene->GetAccelerator()->Intersect(ray, tCurr, is, true))
26         {
27             if (light->IsInfinite())
28             {
29                 return true;
30             }
31             float distanceToLight = (lightPos - ray.GetOrigin()).Length
32             ();
33             float distance = (is.GetPoint() - ray.GetOrigin()).Length()
34             ;
35             return (distance) < (distanceToLight);
36         }
37         return false;
38     }
39
40     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
41                        sampler) const override
42     {
43         Colour colour(0);
44         Intersection intersection;
45         if (scene->Intersect(ray, intersection))
46         {
47             auto lights = scene->GetLights();
48             Point hitpoint = intersection.GetPoint();
```

```

46     Vector Wo(-ray.GetDirection());
47     Material* material = intersection.material;
48
49     for (Light* light : lights)
50     {
51         Vector Wi;
52         Point SampleOnLight;
53         Colour L = light->SampleLight(Point2D(sampler->Sample1D
54             (), sampler->Sample1D()), SampleOnLight, Wi,
55             intersection);
56
57         Colour F = material->F(intersection, Wo, Wi);
58
59         if ((material->GetType() & BRDF::Type::EMISSIVE) ==
60             BRDF::Type::EMISSIVE)
61         {
62             colour += light->Le(ray, intersection);
63
64             if ((material->GetType() & BRDF::Type::DIFFUSE) ==
65                 BRDF::Type::DIFFUSE)
66             {
67                 Ray shadowRay(hitpoint, Wi);
68                 if (!IsInShadow(scene, shadowRay, light,
69                     SampleOnLight))
70                 {
71                     colour += L * F * Max(0.0f, Dot(Wi,
72                         intersection.GetNormal()));
73                 }
74             }
75
76             for (Light* light : scene->GetLights())
77             {
78                 if (light->IsInfinite())
79                 {
80                     colour += light->Le(ray, intersection);
81                 }
82             }
83             return colour;
84
85     }
86
87 };

```

2.6.4 DirectMatIntegrator

Listing 2.55: DirectMatIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Light.h"
6 #include "../Scene.h"
7 #include "../Sample.h"
8
9 class DirectMatIntegrator : public Integrator
10 {
11 public:
12     DirectMatIntegrator()
13     {}
14
15
16     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler* s)
17         const override
18     {
19         Intersection intersection;
20         if (scene->Intersect(ray, intersection))
21         {
22             Colour color(0.0f);
23             Point point = intersection.GetPoint();
24             Normal normal = Normal(intersection.ShadingBasis.W);
25             const Material* bsdf = intersection.material;
26             Vector wo = -ray.GetDirection();
27             float pdf;
28             Vector wi;
29             Colour F = bsdf->SampleF(intersection, wo, wi, pdf);
30             for (Light* emitter : scene->GetLights())
31             {
32                 Ray r(point, Normalize(wi));
33                 if ((bsdf->GetType() & BRDF::Type::EMISSIVE) == BRDF::Type::EMISSIVE)
34                 {
35                     color += emitter->Le(ray, intersection);
36                 }
37
38                 Colour power = emitter->Le(r, intersection);
39                 if (scene->Intersect(r, intersection) && (
40                     DistanceSquared(intersection.GetPoint(),
41                         intersection.GetPoint()) + 1e-6 < wi.LengthSquared
42                         ()))
43                 {
44                     continue;
45                 }
46
47                 if (F != Colour(0, 0, 0) && std::max(0.0f, Dot(wi,
48                     normal)) > 0.0)
```

```
44             color += power * F * std::max(0.0f, Dot(wi, normal))
45             ) / pdf;
46         }
47     }
48     Colour Le;
49
50     for (Light* emitter : scene->GetLights())
51     {
52         if (emitter->IsInfinite())
53         {
54             Le += emitter->Le(ray, intersection);
55         }
56     }
57     return Le;
58 }
59 }
60 }
61 }
```

2.6.5 FunctionIntegrator

Listing 2.56: FunctionIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Light.h"
6 #include "../Scene.h"
7
8 #include <functional>
9
10 class FunctionIntegrator : public Integrator
11 {
12 public:
13     typedef std::function<float(float x, float y)> Function;
14     FunctionIntegrator()
15         : func([](float x, float y) {return x;})
16     {
17     }
18
19     FunctionIntegrator(Function f)
20         : func(f)
21     {
22     }
23
24 }
25
26
27     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
28                         sampler) const override
29     {
30
31         Intersection intersection;
32         float x = (ray.GetDirection().GetX()) * 100.0f;
33         float y = (ray.GetDirection().GetY()) * 100.0f;
34         float z = func(x, y);
35         return Colour(z, z, z);
36     }
37
38 private:
39     Function func = nullptr;
40 };
```

2.6.6 NormalIntegrator

Listing 2.57: NormalIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 //#include "../Materials/DiffuseMaterial.h"
5 //#include "../Materials/PhongMaterial.h"
6 #include "../Material.h"
7 #include "../Light.h"
8 #include "../Scene.h"
9
10 class NormalIntegrator : public Integrator
11 {
12 public:
13     NormalIntegrator()
14     {}
15
16     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
17                         sampler) const override
18     {
19         Intersection intersection;
20         if (!scene->Intersect(ray, intersection))
21         {
22             return Colour(0,0,0);
23         }
24         Normal N = intersection.GetNormal();
25         N = Normal(fabs(N.GetX()), fabs(N.GetY()), fabs(N.GetZ()));
26
27         return Colour(N.GetX(), N.GetY(), N.GetZ());
28     }
29
30 private:
31 };
```

2.6.7 PathTracingIntegrator

Listing 2.58: PathTracingIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Scene.h"
6
7 class PathTracingIntegrator : public Integrator
8 {
9 public:
10     PathTracingIntegrator(int maxDepth)
11         : MaxDepth(maxDepth)
12     {
13     }
14
15     bool IsInShadow(const Scene* scene, const Ray& ray, Light* light,
16                      Point& lightPos) const
17     {
18         Intersection is;
19         Intersection iss;
20         float tCurr = INFINITY;
21         float tMin = INFINITY;
22         bool hit = false;
23         // Need to find a good value otherwise you get speckled shadows
24         float epsilon = 1e-6;
25         if (scene->GetAccelerator()->Intersect(ray, tCurr, is, true))
26         {
27             if (light->IsInfinite())
28             {
29                 return true;
30             }
31
32             float distanceToLight = (lightPos - ray.GetOrigin()).Length
33             ();
34             float distance = (is.GetPoint() - ray.GetOrigin()).Length()
35             ;
36             return (distance) < (distanceToLight);
37         }
38         return false;
39     }
40
41     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
42                        sampler) const override
43     {
44         Colour Lo;
45         Intersection currentIntersection;
```

```

46         return 0.0f;
47     }
48     if (scene->Intersect(ray, currentIntersection))
49     {
50         auto lights = scene->GetLights();
51         Point hitpoint = currentIntersection.GetPoint();
52         Vector Wo(-ray.GetDirection());
53         Material* material = currentIntersection.material;
54
55         for (Light* light : lights)
56         {
57             //if (light->IsInfinite())
58             //    continue;
59             Vector Wi;
60             Point SampleOnLight;
61             Colour L = light->SampleLight(Point2D(sampler->Sample1D
62                 (), sampler->Sample1D()), SampleOnLight, Wi,
63                 currentIntersection);
64             Ray shadowRay(hitpoint, Wi);
65             bool shadow = IsInShadow(scene, shadowRay, light,
66                 SampleOnLight);
67
68             if (material->IsTranslucent())
69             {
70                 if (!shadow)
71                 {
72                     Lo += L * material->F(currentIntersection, Wo,
73                         Wi) * fabsf(Dot(Wi, currentIntersection.
74                             GetNormal()));
75                 }
76             }
77
78             if ((material->GetType() & BRDF::Type::DIFFUSE) == BRDF
79                 ::Type::DIFFUSE)
80             {
81                 if (!shadow)
82                 {
83                     Colour F = material->F(currentIntersection, Wo,
84                         Wi);
85                     Lo += L * F * Max(0.0f, Dot(Wi,
86                         currentIntersection.GetNormal())));
87                 }
88             }
89         }
90     }
91 }
```

```

86
87         }
88
89     }
90     Vector wi;
91     float pdf;
92     Colour F = material->SampleF(currentIntersection, Wo, wi,
93                                     pdf);
94     ray.Depth++;
95     if ((material->GetType() & BRDF::Type::EMISSIVE) == BRDF::
96         Type::EMISSIVE)
97     {
98         Ray pathRay = Ray(currentIntersection.GetPoint(),
99                             Sample::CosineHemisphere(sampler->Sample2D()));
100        pathRay.Depth = ray.Depth;
101        return F + Li(scene, pathRay, sampler) * fabsf(Dot(wi,
102                                                       currentIntersection.GetNormal()));
103    }
104    if (F == Colour(0) || pdf == 0.0f)
105    {
106        return 0.0f;
107    }
108    if (ray.GetDepth() < MaxDepth)
109    {
110        if ((material->GetType() & BRDF::Type::SPECULAR) ==
111            BRDF::Type::SPECULAR)
112        {
113            Lo += LiSpecular(scene, sampler, ray,
114                              currentIntersection, currentIntersection.
115                              material);
116        }
117        Ray pathRay = Ray(currentIntersection.GetPoint(), wi.
118                            Hat());
119        pathRay.Depth = ray.Depth;
120        if ((material->GetType() & BRDF::Type::REFRACTION) ==
121            BRDF::Type::REFRACTION)
122        {
123            Lo += LiRefractive(scene, sampler, ray,
124                               currentIntersection, currentIntersection.
125                               material);
126        }
127        //if ((material->GetType() & BRDF::Type::GLOSSY) ==
128            BRDF::Type::GLOSSY)
129        {
130            Lo += F * Li(scene, pathRay, sampler) * fabsf(Dot(
131                                                       wi, currentIntersection.GetNormal())) / pdf;
132        }
133    }
134 }

```

```

126     else
127     {
128         //Colour Lo;
129         for (Light* light : scene->GetLights())
130         {
131             if (light->IsInfinite())
132             {
133                 Lo += light->Le(ray, currentIntersection);
134             }
135         }
136         return Lo == Colour(0, 0, 0) ? Colour(0, 0, 0) : Lo;
137     }
138
139     return Lo;
140 }
141
142 Colour LiSpecular(const Scene* scene, Sampler* sampler, const Ray&
143 ray, Intersection& intersection, Material* brdf) const
144 {
145     Vector n(intersection.GetNormal());
146     Vector wo = -ray.GetDirection();
147     Vector wi;
148     float pdf;
149     Colour F = brdf->SampleF(intersection, wo, wi, pdf);
150     Colour L;
151     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) != 0.0f)
152     {
153         Ray ReflectedRay(intersection.GetPoint(), wi.Hat());
154         ReflectedRay.Depth = ray.GetDepth();
155         Colour li = Li(scene, ReflectedRay, sampler) ;
156         L = F * li * Max(0.0f, Dot(wi, n));
157     }
158
159     return L;
160 }
161
162 Colour LiRefractive(const Scene* scene, Sampler* sampler, const Ray
163 & ray, Intersection& intersection, Material* brdf) const
164 {
165     Vector n(intersection.GetNormal());
166     Vector wo = -ray.GetDirection();
167     Vector wi;
168     float pdf;
169     Colour F = intersection.material->SampleF(intersection, wo, wi,
170         pdf);
171     Colour L;
172     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) != 0.0f)
173     {
174         Ray RefractedRay(intersection.GetPoint(), wi);
175         RefractedRay.Depth = ray.GetDepth();
176         Colour li = Li(scene, RefractedRay, sampler);
177         L = F * li * fabsf(Dot(wi, n));

```

```
175         }
176         return L;
177     }
178
179
180 private:
181     int MaxDepth;
182 }
```

2.6.8 PhotonMappingIntegrator

Listing 2.59: PhotonMappingIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Scene.h"
6 #include "../Sample.h"
7 #include "../Photon.h"
8 #include "../../Core/Math/Statistics.h"
9
10 class PhotonMappingIntegrator : public Integrator
11 {
12 public:
13     PhotonMappingIntegrator(int maxPhotons, int maxCausticPhotons,
14                             float photonGatherRadius)
15         : MaxPhoton(maxPhotons),
16           MaxCausticPhotons(maxCausticPhotons),
17           PhotonGatherRadius(photonGatherRadius),
18           MaxDepth(5)
19     {}
20
21     void Preprocess(Scene* scene, Sampler* sampler) override
22     {
23         PhotonEmission(scene, sampler);
24         CausticPhotonEmission(scene, sampler);
25     }
26
27     void PhotonEmission(const Scene* scene, Sampler* sampler)
28     {
29         std::vector<Photon> directPhotons;
30         std::vector<float> lightPower;
31         for (Light* light: scene->GetLights())
32         {
33             lightPower.push_back(light->GetPower().Average());
34         }
35         Distribution1D lightDistribution(lightPower);
36         //std::ofstream output("global.photon");
37         //std::ofstream output("global.photon", ios_base::out |
38         //    ios_base::binary);
39
40         int number0fEmittedPhotons = 0;
41         for (int i = 0; i < MaxPhoton; ++i)
42         {
43             Ray photonRay;
44             float x;
45             float pdf;
46             lightDistribution.Sample1D(sampler->Sample1D(), x, pdf);
47             int lightIndex = x * (scene->GetLights().size() - 1);
48             Light* light = scene->GetLights()[lightIndex];
49             Normal normal;
```

```

48     Colour lp = light->SampleLight(scene, sampler, photonRay,
49         normal);
50     Intersection intersection;
51     Photon photon;
52     photon.position = intersection.GetPoint();
53     photon.incidentDirection = photonRay.GetDirection();
54     photon.power = lp / pdf;
55
56     while (scene->Intersect(photonRay, intersection))
57     {
58         Material* material = intersection.material;
59
60         if ((material->GetType() & BRDF::Type::DIFFUSE) ==
61             BRDF::Type::DIFFUSE)
62         {
63             Vector scatterDirection = Sample::UniformSphere
64                 (sampler->Sample2D());
65             Colour diffuse = material->F(intersection,
66                 scatterDirection, photonRay.GetDirection())
67                 ;
68             Colour p = material->Rho(scatterDirection,
69                 intersection);
70
71             // Use russian roulette to determine photon
72             // scattering
73             double probabilityOfReflection = p.Clamp().
74                 Average();
75             float u = sampler->Sample1D();
76             ++numberOfEmittedPhotons;
77             photon.position = intersection.GetPoint();
78             photon.power *= diffuse /
79                 probabilityOfReflection;
80             PhotonMap.push_back(photon);
81             if (u < probabilityOfReflection)
82             {
83
84                 photonRay = Ray(intersection.GetPoint(),
85                     scatterDirection);
86             }
87             else
88             {
89                 break;
90             }
91         }
92         else if ((material->GetType() & BRDF::Type::
93             REFRACTION) == BRDF::Type::REFRACTION)
94         {
95             Vector wi;
96             Vector wo = -photonRay.GetDirection();
97             float specularPDF;
98             Colour F = intersection.material->SampleF(
99                 intersection, wo, wi, specularPDF);

```

```

88         float probabilityOfRefraction = intersection.
89             material->Rho(wo, intersection).Average();
90         float u = sampler->Sample1D();
91         photon.position = intersection.GetPoint();
92         photon.incidentDirection = photonRay.
93             GetDirection();
94         photon.power = light->GetPower();
95         if (u < probabilityOfRefraction)
96         {
97             photonRay = Ray(intersection.GetPoint(), wi
98                             );
99         }
100        else
101        {
102            break;
103        }
104    }
105    else if ((material->GetType() & BRDF::Type:::
106               EMISSIVE) == BRDF::Type::EMISSIVE)
107    {
108        break;
109    }
110    PhotonMap.scalePhotonPw(1 / float(numberOfEmittedPhotons), 0,
111                             PhotonMap.size());
112    PhotonMap.build();
113    //for (Photon& photon: PhotonMap.photons)
114    //  output << photon;
115    //output.close();
116
117}
118
119 void CausticPhotonEmission(const Scene* scene, Sampler* sampler)
120 {
121     if (scene->GetSpecularGeometry().size() != 0)
122     {
123         std::vector<float> lightPower;
124         for (Light* light : scene->GetLights())
125         {
126             lightPower.push_back(light->GetPower().Average());
127         }
128         Distribution1D lightDistribution(lightPower);
129
130         int numberOfCausticPhotons = 0;
131
132         //std::ofstream output("caustic.photon");
133         //std::ofstream output("caustic.photon", ios_base::out |
134             ios_base::binary);

```

```

134     while (numberOfCausticPhotons < MaxCausticPhotons)
135     {
136         float x;
137         float pdf;
138         lightDistribution.Sample1D(sampler->Sample1D(), x, pdf)
139             ;
140         int lightIndex = x * (scene->GetLights().size() - 1);
141         Light* light = scene->GetLights()[lightIndex];
142         Ray photonRay;
143         Normal photonNormal;
144         Colour lightIrradiance = light->SampleLight(scene,
145             sampler, photonRay, photonNormal);
146         for (Primitive* specularSurface : scene->
147             GetSpecularGeometry())
148         {
149             float tMin = INFINITE;
150             float epsilon = 1e-6;
151             Intersection specularIntersection;
152             if (specularSurface->IntersectRay(photonRay, tMin,
153                 epsilon, specularIntersection))
154             {
155                 if ((specularIntersection.material->GetType() &
156                     BRDF::Type::REFRACTION) == BRDF::Type::
157                     REFRACTION)
158                 {
159                     Vector wi;
160                     Vector wo = -photonRay.GetDirection();
161                     float specularPDF;
162                     Colour F = specularIntersection.material->
163                         SampleF(specularIntersection, wo, wi,
164                             specularPDF);
165                     float probabilityOfRefraction =
166                         specularIntersection.material->Rho(wo,
167                             specularIntersection).Average();
168                     if (F == Colour(0,0,0) ||
169                         probabilityOfRefraction == 0.0f)
170                     {
171                         continue;
172                     }
173                     float u = sampler->Sample1D();
174                     Photon photon;
175                     photon.position = specularIntersection.
176                         GetPoint();
177                     photon.incidentDirection = photonRay.
178                         GetDirection();
179                     photon.power = light->GetPower();
180                     if (u < probabilityOfRefraction)
181                     {
182                         Ray pp = Ray(specularIntersection.
183                             GetPoint(), wi);
184                         TracePhoton(scene, pp,
185                             specularIntersection, photon,

```

```

                                sampler, number0fCausticPhotons);
171                         }
172                     else
173                     {
174                         break;
175                     }
176                 }else
177                 {
178                     break;
179                 }
180             }
181         }
182     }
183 }
184 CausticPhotonMap.scalePhotonPw(1 / float(
185     number0fCausticPhotons), 0, CausticPhotonMap.size());
186 CausticPhotonMap.build();
187 //for (Photon& photon : CausticPhotonMap.photons)
188 //    output << photon;
189 //output.close();
190 }
191 else
192 {
193     std::cerr << "No specular object: Cannot create caustic
194         photon map";
195 }
196 }
197
198 void TracePhoton(const Scene* scene, Ray& ray, Intersection&
199 intersection, Photon photon, Sampler* sampler, int&
200 number0fCausticPhotons)
201 {
202     int bounces = 0;
203     while (scene->Intersect(ray, intersection) || bounces < 10)
204     {
205         bounces++;
206         Material* material = intersection.material;
207         if ((material->GetType() & BRDF::Type::DIFFUSE) == BRDF::
208             Type::DIFFUSE)
209         {
210             Photon p;
211             p.position = intersection.GetPoint();
212             p.incidentDirection = ray.GetDirection();
213             photon.power *= material->F(intersection, -ray.
214                 GetDirection(), photon.incidentDirection) ;
215             p.power = photon.power;
216             CausticPhotonMap.push_back(p);
217             number0fCausticPhotons++;
218             break;
219         }
220     }

```

```

216     if ((material->GetType() & BRDF::Type::REFRACTION) == BRDF
217         ::Type::REFRACTION)
218     {
219         Vector wi;
220         Vector wo = -ray.GetDirection();
221         float pdf;
222         Colour F = material->SampleF(intersection, wo, wi, pdf)
223             ;
224         float probabilityOfRefraction = material->Rho(wo,
225             intersection).Average();
226         float u = sampler->Sample1D();
227
228         if (F == Colour(0.0f) || pdf == 0.0f)
229         {
230             break;
231         }
232         if (u > probabilityOfRefraction)
233         {
234             ray = Ray(intersection.GetPoint(), wi);
235
236             //photon.power *= F;// / 10.0f;
237         }
238         else
239         {
240             //bounces++;
241             //ray = Ray(intersection.GetPoint(), Reflect(-ray.
242                 GetDirection(), intersection.GetNormal()));
243             //photon.power *= F / 10.0f;
244             break;
245         }
246     }
247     Colour IrradianceEstimate(const KDTree& tree, KDTreeQuery& query,
248         const Ray& ray, Intersection& intersection, bool useCone = true
249     ) const
250     {
251         Colour irradiance;
252         tree.findNN(query);
253         std::vector<KDTreeResult> results = query.result;
254         if (results.size() < 8)
255         {
256             return irradiance;
257         }
258         float maxDistanceToPhoton = -1e10;
259         for (KDTreeResult& result : results)
260         {
261             if (fabsf(result.distanceSquared) > maxDistanceToPhoton)
262                 maxDistanceToPhoton = fabsf(result.distanceSquared);
263         }

```

```

262
263     //distanceToPhoton = query.searchDistanceSquared;
264
265     for (KDTreeResult& result: results)
266     {
267         Photon& photon = result.photon;
268         //if (Dot(photon.incidentDirection, intersection.GetNormal()
269         //() < 0.0f)
270         {
271             if (useCone)
272             {
273                 float distanceToPhoton = sqrtf(fabsf(result.
274                     distanceSquared));
275                 float maxDist = sqrtf(maxDistanceToPhoton);
276                 irradiance += photon.power * fabsf(Dot(-photon.
277                     incidentDirection, intersection.GetNormal())) *
278                     ConeFilter(distanceToPhoton, maxDist);
279             }
280             else
281             {
282                 float distanceToPhoton = fabsf(result.
283                     distanceSquared);
284                 irradiance += photon.power * fabsf(Dot(-photon.
285                     incidentDirection, intersection.GetNormal())) *
286                     GaussianFilter(distanceToPhoton,
287                     maxDistanceToPhoton);
288             }
289         }
290         float densityEstimate = 1.0f / (M_PI * maxDistanceToPhoton);
291         irradiance *= densityEstimate;
292
293         return irradiance;
294     }
295
296     float GaussianFilter(float searchDistanceSquared, float
297     maxPhotonDistanceSquared, float alpha = 1.818f, float beta =
298     1.953f) const
299     {
300         float b = (1 - expf(-beta * (searchDistanceSquared / (2 *
301             maxPhotonDistanceSquared))));
302         float c = 1 - expf(-beta);
303         return alpha * (1 - b / c);
304     }
305
306     float ConeFilter(float distanceToPhoton, float
307     maximumDistanceToPhotons, float k = 1.1) const
308     {
309         float f = 1 - (distanceToPhoton / (k * maximumDistanceToPhotons
310             ));
311         float c = 1 - 2 / (3 * k);

```

```

301     return f / c;
302 }
303
304 bool IsInShadow(const Scene* scene, const Ray& ray, Light* light,
305                  Point& lightPos) const
306 {
307     Intersection is;
308     float tCurr = INFINITY;
309     if (scene->GetAccelerator()->Intersect(ray, tCurr, is, true))
310     {
311         if (light->IsInfinite())
312         {
313             return true;
314         }
315         float distanceToLight = (lightPos - ray.GetOrigin()).Length
316         ();
317         float distance = (is.GetPoint() - ray.GetOrigin()).Length()
318         ;
319         return (distance) < (distanceToLight);
320     }
321     return false;
322 }
323
324 virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
325                      sampler) const override
326 {
327     Intersection intersection;
328     Colour colour;
329     if (scene->Intersect(ray, intersection))
330     {
331
332         Point hitpoint = intersection.GetPoint();
333         Normal N = intersection.GetNormal();
334
335         // direct lighting
336         auto lights = scene->GetLights();
337         Vector Wo(-ray.GetDirection());
338         Material* material = intersection.material;
339
340         for (Light* light : lights)
341         {
342             Vector Wi;
343             Point SampleOnLight;
344             Colour L = light->SampleLight(Point2D(sampler->Sample1D
345             (), sampler->Sample1D()), SampleOnLight, Wi,
346             intersection);
347             Colour F = material->F(intersection, Wo, Wi);
348
349             if ((material->GetType() & BRDF::Type::EMISSIVE) ==
350                 BRDF::Type::EMISSIVE)

```

```

346
347         colour += light->Le(ray, intersection);
348     }
349
350     if (material->IsTranslucent())
351     {
352         colour += L * F;// *fabsf(Dot(Wi, intersection.
353                                     GetNormal()));
354     }
355     if ((material->GetType() & BRDF::Type::DIFFUSE) == BRDF
356         ::Type::DIFFUSE)
357     {
358         Ray shadowRay(hitpoint, Wi);
359         if (!IsInShadow(scene, shadowRay, light,
360                         SampleOnLight))
361         {
362             colour += L * F * fabsf(Dot(Wi, intersection.
363                                         GetNormal()));
364         }
365     }
366     ray.Depth++;
367     if (ray.GetDepth() < MaxDepth)
368     {
369         if ((material->GetType() & BRDF::Type::SPECULAR) ==
370             BRDF::Type::SPECULAR)
371         {
372             colour += LiSpecular(scene, sampler, ray,
373                                   intersection, intersection.material);
374         }
375         if ((material->GetType() & BRDF::Type::REFRACTION) ==
376             BRDF::Type::REFRACTION)
377         {
378             colour += LiRefractive(scene, sampler, ray,
379                                   intersection, intersection.material);
380         }
381     }
382
383 //Vector wi;
384 //float pdf;
385 //Colour F = intersection.material->SampleF(intersection,
386 //                                              ray.GetDirection(), wi, pdf);
387 //Ray r(hitpoint, wi);
388 //r.Depth = ray.Depth;
389 //if (scene->Intersect(r, intersection))
390 //if (r.Depth < MaxDepth)
391 //{

```

```

389         // KDTTreeQuery query;
390         // query.queryPosition = intersection.GetPoint();
391         // query.maxNumber = 50;
392         // query.searchDistanceSquared = PhotonGatherRadius *
393         // PhotonGatherRadius;
394         // colour += Li(scene, r, sampler) + IrradianceEstimate(
395         // PhotonMap, query, r, intersection) * F;
396         // //colour += IrradianceEstimate(PhotonMap, query, ray,
397         // intersection) * F;
398     //}
399
400 #if 0
401     Vector wi;
402     float pdf;
403     Colour F = intersection.material->SampleF(intersection, -
404         ray.GetDirection(), wi, pdf);
405     Ray r(hitpoint, wi);
406     r.Depth = ray.Depth;
407     //if (scene->Intersect(r, intersection))
408     {
409         KDTTreeQuery query;
410         query.queryPosition = intersection.GetPoint();
411         query.maxNumber = 50;
412         query.searchDistanceSquared = PhotonGatherRadius *
413         PhotonGatherRadius;
414         //colour += Li(scene, r, sampler) + IrradianceEstimate(
415         // query, r, intersection);
416         colour += IrradianceEstimate(PhotonMap, query, ray,
417             intersection);
418     }
419 #endif
420
421     Vector wi;
422     float pdf;
423     Colour F = intersection.material->SampleF(intersection, -
424         ray.GetDirection(), wi, pdf);
425
426     // Caustics
427     if (CausticPhotonMap.size() != 0)
428     {
429         KDTTreeQuery causticQuery;
430         causticQuery.queryPosition = intersection.GetPoint();
431         causticQuery.maxNumber = 50;
432         causticQuery.searchDistanceSquared = 9;
433         colour += IrradianceEstimate(CausticPhotonMap,
434             causticQuery, ray, intersection) * F;
435     }
436
437     Ray r(hitpoint, wi);
438     r.Depth = ray.Depth;

```

```

431     if (scene->Intersect(r, intersection) && !BRDF::IsBSDFType(
432         material->GetType(), BRDF::Type::REFRACTION))
433         //if (!BRDF::IsBSDFType(material->GetType(), BRDF::Type::
434             REFRACTION))
435     {
436         KDTTreeQuery query;
437         query.queryPosition = intersection.GetPoint();
438         query.maxNumber = 50;
439         query.searchDistanceSquared = PhotonGatherRadius *
440             PhotonGatherRadius;
441         //colour += (Li(scene, r, sampler) + IrradianceEstimate(
442             PhotonMap, query, r, intersection)) * F;
443         colour += IrradianceEstimate(PhotonMap, query, r,
444             intersection) * F;
445     }
446     return Colour(0.0f, 0.0f, 0.0f);
447 }
448
449 Colour LiSpecular(const Scene* scene, Sampler* sampler, const Ray&
450 ray, Intersection& intersection, Material* brdf) const
451 {
452     Vector n(intersection.GetNormal());
453     Vector wo = -ray.GetDirection();
454     Vector wi;
455     float pdf;
456     Colour F = brdf->SampleF(intersection, wo, wi, pdf);
457     Colour L;
458     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) != 0.0f)
459     {
460         Ray ReflectedRay(intersection.GetPoint(), wi.Hat());
461         ReflectedRay.Depth = ray.GetDepth();
462         Colour li = Li(scene, ReflectedRay, sampler);
463         L = F * li * fabsf(Dot(wi, n));
464     }
465     return L;
466 }
467
468 Colour LiRefractive(const Scene* scene, Sampler* sampler, const Ray
469 & ray, Intersection& intersection, Material* brdf) const
470 {
471     Vector n(intersection.GetNormal());
472     Vector wo = -ray.GetDirection();
473     Vector wi;
474     float pdf;
475     Colour F = intersection.material->SampleF(intersection, wo, wi,
476         pdf);

```

```

475     Colour L;
476     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) > 0.0f)
477     {
478         Ray RefractedRay(intersection.GetPoint(), wi);
479         RefractedRay.Depth = ray.GetDepth();
480         Colour li = Li(scene, RefractedRay, sampler);
481         L = F * li * fabsf(Dot(wi, n));
482     }
483
484     return L;
485 }
486
487 private:
488     int MaxPhoton;
489     float PhotonGatherRadius;
490     KDTree PhotonMap;
491     KDTree CausticPhotonMap;
492     int MaxDepth;
493     int MaxCausticPhotons;
494 };

```

2.6.9 WhittedIntegrator

Listing 2.60: WhittedIntegrator header file

```
1 #pragma once
2
3 #include "../Integrator.h"
4 #include "../Material.h"
5 #include "../Scene.h"
6 #include "../Volume/Volume.h"
7
8 #define WHITTED_REFLECTION      1
9 #define WHITTED_REFRACTION      1
10 #define WHITTED_SHADOW          1
11 #define WHITTED_DEBUG_SHADOWS   0
12 #define WHITTED_REFRACTION_AFFECTS_SHADOW 1
13
14 class WhittedIntegrator : public Integrator
15 {
16 public:
17     WhittedIntegrator(int maxDepth)
18         : MaxDepth(maxDepth)
19     {}
20
21     bool IsInShadow(const Scene* scene, const Ray& ray, Light* light,
22                      Point& lightPos) const
23     {
24         Intersection is;
25         Intersection iss;
26         float tCurr = INFINITY;
27         float tMin = INFINITY;
28         bool hit = false;
29         // Need to find a good value otherwise you get speckled shadows
30         float epsilon = 1e-6;
31         if (scene->GetAccelerator()->Intersect(ray, tCurr, is, true))
32         {
33             if (light->IsInfinite())
34             {
35                 return true;
36             }
37             float distanceToLight = (lightPos - ray.GetOrigin()).Length
38             ();
39             float distance = (is.GetPoint() - ray.GetOrigin()).Length()
40             ;
41             return (distance) < (distanceToLight);
42         }
43         return false;
44     }
45     virtual Colour Li(const Scene* scene, const Ray& ray, Sampler*
46                        sampler) const override
```

```

46 Colour colour;
47 Intersection intersection;
48 if (scene->Intersect(ray, intersection))
49 {
50     auto lights = scene->GetLights();
51     Point hitpoint = intersection.GetPoint();
52     Vector Wo(-ray.GetDirection());
53     Material* material = intersection.material;
54
55
56     for (Light* light : lights)
57     {
58         Vector Wi;
59         Point SampleOnLight;
60         Colour L = light->SampleLight(Point2D(sampler->Sample1D
61             (), sampler->Sample1D()), SampleOnLight, Wi,
62             intersection);
63         Colour F = material->F(intersection, Wo, Wi);
64
65         if ((material->GetType() & BRDF::Type::EMISSIVE) ==
66             BRDF::Type::EMISSIVE)
67         {
68             colour += light->Le(ray, intersection);
69         }
70
71         if (material->IsTranslucent())
72         {
73             //std::cout << "Hello";
74             colour += L * F;// *fabsf(Dot(Wi, intersection.
75                 GetNormal()));
76
77         }
78         if ((material->GetType() & BRDF::Type::DIFFUSE) == BRDF
79             ::Type::DIFFUSE)
80         {
81             if (!ray.IsRefractiveRay())
82             {
83
84                 #if WHITTED_SHADOW
85                     Ray shadowRay(hitpoint, Wi, 0, (SampleOnLight -
86                         hitpoint).Length());
87                     if (!IsInShadow(scene, shadowRay, light,
88                         SampleOnLight))
89                     {
90                         if (scene->SVolumeIntegrator == nullptr)
91                         {
92
93                             colour += L * F * Max(0.0f, Dot(Wi,
94                                 intersection.GetNormal()));
95                         }
96                     }
97                 }
98             }
99         }

```

```

90
91     Colour transmittance = scene->
92         SVolumeIntegrator->Transmittance(
93             scene, shadowRay, sampler);
94         //colour += (transmittance * L * F *
95         //scene->GetVolume()->
96         //GetPhaseFunction()->PDF(Wo, Wi));
97         colour += transmittance * L * F * fabsf
98             (Dot(Wi, intersection.GetNormal())));
99             ;
100
101         }
102     }
103     else
104     {
105         colour += F * Max(0.0f, Dot(Wi, intersection.
106             GetNormal())));
107     }
108 #if WHITTED_DEBUG_SHADOWS
109     else
110     {
111         colour += Colour(0,1,0);
112     }
113     #endif
114
115     #else
116         colour += L * F * fabsf(Dot(Wi, intersection.
117             GetNormal()));// * texture->GetValue(is.U, is.V
118             , is);
119         #endif
120     }
121 }
122
123 // Calculate perfect specular reflection and transmission
124 ray.Depth += 1;
125 if (ray.GetDepth() < MaxDepth)
126 {
127 #if WHITTED_REFLECTION
128     if ((material->GetType() & BRDF::Type::SPECULAR) ==
129         BRDF::Type::SPECULAR)
130     {
131         colour += LiSpecular(scene, sampler, ray,
132             intersection, intersection.material);
133     }
134     #endif
135
136 #if WHITTED_REFRACTION
137     if ((material->GetType() & BRDF::Type::REFRACTION) ==
138         BRDF::Type::REFRACTION)
139     {
140         colour += LiRefractive(scene, sampler, ray,
141             intersection, intersection.material);
142     }
143     #else
144     #endif

```

```

129
130     //if ((material->GetType() & BRDF::Type::GLOSSY) ==
131     //    BRDF::Type::GLOSSY)
132     {
133         Vector wi;
134         float pdf;
135         Colour F = material->SampleF(intersection, Wo, wi,
136                                         pdf);
137         if (F != Colour(0, 0, 0) && fabsf(Dot(wi,
138                                         intersection.GetNormal())) > 0.0f)
139         {
140             Ray glossyRay(intersection.GetPoint(), wi);
141             glossyRay.Depth = ray.Depth;
142             colour += F * Li(scene, glossyRay, sampler) *
143                         fabsf(Dot(wi, intersection.GetNormal())) /
144                         pdf;
145         }
146     }
147 }
148
149     if (scene->SVolumeIntegrator != nullptr)
150     {
151         auto a = Sample::UniformSphere(sampler->Sample2D());
152         if (Dot(a, intersection.GetNormal()) < 0.0f)
153         {
154             a = -a;
155         }
156         Ray rr(hitpoint, a);
157         rr.Depth = ray.Depth;
158         colour += scene->SVolumeIntegrator->Li(scene, rr,
159                                         sampler).Luminance;
160     }
161
162     return colour;
163 }
164 else
165 {
166     Colour Le;
167     for (Light* light : scene->GetLights())
168     {
169         if (light->IsInfinite())
170         {
171             Le += light->Le(ray, intersection);
172         }
173     }
174     return Le == Colour(0, 0, 0) ? Colour(0,0,0) : Le;
175 }
176
177 Colour LiSpecular(const Scene* scene, Sampler* sampler, const Ray&
178                     ray, Intersection& intersection, Material* brdf) const
179 {

```

```

174     Vector n(intersection.GetNormal());
175     Vector wo = -ray.GetDirection();
176     Vector wi;
177     float pdf;
178     Colour F = brdf->SampleF(intersection, wo, wi, pdf);
179     Colour L;
180     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) != 0.0f)
181     {
182         Ray ReflectedRay(intersection.GetPoint(), wi.Hat());
183         ReflectedRay.Depth = ray.GetDepth();
184         Colour li = Li(scene, ReflectedRay, sampler);
185         L = F * li * fabsf(Dot(wi, n));
186     }
187
188     return L;
189 }
190
191 Colour LiRefractive(const Scene* scene, Sampler* sampler, const Ray
192   & ray, Intersection& intersection, Material* brdf) const
193 {
194     Vector n(intersection.GetNormal());
195     Vector wo = -ray.GetDirection();
196     Vector wi;
197     float pdf;
198     Colour F = intersection.material->SampleF(intersection, wo, wi,
199       pdf);
200     Colour L;
201     if (F != Colour(0, 0, 0) && fabsf(Dot(wi, n)) != 0.0f)
202     {
203         Ray RefractedRay(intersection.GetPoint(), wi.Hat());
204
205         RefractedRay.Depth = ray.GetDepth();
206         Colour li = Li(scene, RefractedRay, sampler);
207         L = F * li * fabsf(Dot(wi, n));
208     }
209
210     return L;
211 }
212
213 private:
214     int MaxDepth;
215 };

```

2.7 Light

2.7.1 AreaLight

Listing 2.61: AreaLight header file

```
1 #pragma once
2
3 #include "../Light.h"
4
5 class AreaLight : public Light
6 {
7 public:
8     AreaLight(Transform& localToWorld);
9     virtual Colour L(const Point& p, const Normal& n, const Vector& wo,
10                      Intersection& intersection) const = 0;
11
12     bool IsArea() const override;
13 };
14 inline AreaLight::AreaLight(Transform& localToWorld)
15     : Light(localToWorld)
16 {
17 }
18
19 inline bool AreaLight::IsArea() const
20 {
21     return true;
22 }
```

2.7.2 DiffuseAreaLight

Listing 2.62: DiffuseAreaLight header file

```
1 #pragma once
2
3 #include "AreaLight.h"
4 #include "../Geometry.h"
5 #include "../Material.h"
6 #include "../Primitive.h"
7 #include "../../Core/Math/Point2D.h"
8 #include "../../Core/Math/Normal.h"
9 #include "../Primitives/GeometricPrimitive.h"
10
11 class DiffuseAreaLight : public AreaLight, public GeometricPrimitive
12 {
13 public:
14
15     DiffuseAreaLight(Transform& localToWorld, const Colour& m_radiance,
16                       Geometry* m_geometry, Material* m_material)
17         : AreaLight(localToWorld),
18           GeometricPrimitive(m_geometry, m_material),
19           M_Radiance(m_radiance)
20     {
21
22
23     Colour SampleLight(const Scene* scene, Sampler* sampler, Ray& ray,
24                         Normal& normal) const override;
25     Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
26                         Intersection& intersection) const override;
27     //Colour Li(Vector& Wi, Intersection& intersect) const override;
28
29     Colour Li(Vector& Wi, Intersection& intersect) const override;
30     Colour Le(const Ray& ray, Intersection& intersection) const
31             override;
32     Colour L(const Point& p, const Normal& n, const Vector& wo,
33               Intersection& intersection) const override;
34
35     bool IsArea() const override;
36     Colour GetPower() const override;
37     Point GetPosition() const override;
38
39 private:
40     Colour M_Radiance;
41 };
42
43 inline Colour DiffuseAreaLight::SampleLight(const Scene* scene, Sampler
44 * sampler, Ray& ray, Normal& normal) const
45 {
46     Normal lightNormal;
47     Point origin = PGeometry->Sample(sampler->Sample2D(), lightNormal);
48     Vector dir = Sample::UniformSphere(sampler->Sample2D());
```

```

44     if (Dot(dir, lightNormal) < 0.0f)
45     {
46         dir = -dir;
47     }
48
49     ray = Ray(origin, dir);
50     float pdf = PGeometry->Pdf(origin);
51     return M_Radiance / pdf;// L(origin, lightNormal, dir, intersection
52     ) / pdf;
53 }
54 inline Colour DiffuseAreaLight::SampleLight(const Point2D& sample,
55     Point& point, Vector& wi, Intersection& intersection) const
56 {
57     // TODO: DiffuseAreaLight::SampleLight must be implemented.
58     Normal lightNormal;
59     point = PGeometry->Sample(Point2D(sample[0], sample[1]),
60         lightNormal);
61     wi = (point - intersection.GetPoint()).Hat();
62     // Convert to solid angle measure
63     float pdf = DistanceSquared(point, intersection.GetPoint()) / (
64         fabsf(Dot(lightNormal, -wi) * PGeometry->SurfaceArea()));
65     return L(point, lightNormal, -wi, intersection) / pdf;
66 }
67 inline Colour DiffuseAreaLight::Li(Vector& Wi, Intersection& intersect)
68     const
69 {
70     return 0.0f;
71 }
72 inline Colour DiffuseAreaLight::Le(const Ray& ray, Intersection&
73     intersection) const
74 {
75     return PMaterial->F(intersection, ray.GetDirection(), (intersection
76         .GetPoint()-ray.GetOrigin()));
77 }
78 inline Colour DiffuseAreaLight::L(const Point& p, const Normal& n,
79     const Vector& wo, Intersection& intersection) const
80 {
81     bool IsOnEmittanceSide = Dot(n, wo) > 0.0f;
82     if (IsOnEmittanceSide)
83         return M_Radiance * PMaterial->F(intersection, wo, (p -
84             intersection.GetPoint()).Hat());
85     return Colour(0.0f);
86 }
87 inline bool DiffuseAreaLight::IsArea() const
88 {

```

```
87         return true;
88     }
89
90     inline Colour DiffuseAreaLight::GetPower() const
91 {
92         return M_Radiance * PGeometry->SurfaceArea() * M_PI;
93     }
94
95     inline Point DiffuseAreaLight::GetPosition() const
96 {
97         Normal normal;
98         Point point = PGeometry->Sample(Point2D(rand() / float(RAND_MAX),
99                                         rand() / float(RAND_MAX)), normal);
100        return point;
101 }
```

2.7.3 DirectedAreaLight

Listing 2.63: DirectedAreaLight header file

```
1 #pragma once
2
3 #include "AreaLight.h"
4 #include "../Geometry.h"
5 #include "../Material.h"
6 #include "../Primitive.h"
7 #include "../../Core/Math/Point2D.h"
8 #include "../../Core/Math/Normal.h"
9 #include "../Primitives/GeometricPrimitive.h"
10
11 class DirectedAreaLight : public AreaLight, public GeometricPrimitive
12 {
13 public:
14
15     DirectedAreaLight(Transform& localToWorld, const Colour& m_radiance
16                 , Geometry* m_geometry, Material* m_material, float
17                 directionalAmount = 0.0f)
18         : AreaLight(localToWorld),
19             GeometricPrimitive(m_geometry, m_material),
20             M_Radiance(m_radiance),
21             M_DirectionAmount(clamp(directionalAmount, 0.0f, 1.0f))
22     {
23     }
24
25     Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
26                         Intersection& intersection) const override;
27 //Colour Li(Vector& Wi, Intersection& intersect) const override;
28
29     Colour Li(Vector& Wi, Intersection& intersect) const override;
30     Colour Le(const Ray& ray, Intersection& intersection) const
31             override;
32     Colour L(const Point& p, const Normal& n, const Vector& wo,
33               Intersection& intersection) const override;
34
35     bool IsArea() const override;
36     Colour GetPower() const override;
37     Point GetPosition() const override;
38 }
39
40 inline Colour DirectedAreaLight::SampleLight(const Point2D& sample,
41                                              Point& point, Vector& wi, Intersection& intersection) const
42 {
43     // TODO: DiffuseAreaLight::SampleLight must be implemented.
44     Normal lightNormal;
```

```

44     point = PGeometry->Sample(Point2D(sample[0], sample[1]),
45         lightNormal);
46     wi = (point - intersection.GetPoint()).Hat();
47
48     // Convert to solid angle measure
49     float t = cosf(DegToRad(M_DirectionAmount * 180.0f)) * 0.5 + 0.5;
50     Vector interp = Vector(-lightNormal * (1 - t) + wi * t).Hatt();
51     Ray r(intersection.GetPoint(), interp);
52
53     float pdf = DistanceSquared(point, intersection.GetPoint()) / (
54         fabsf(Dot(-interp, lightNormal) * PGeometry->SurfaceArea()));
55
56     float tMin = 1e9;
57     float epsilon = 1e-6;
58     Intersection is;
59     if (PGeometry->IntersectRay(r, tMin, epsilon, is))
60     {
61         return L(point, lightNormal, -interp, is) / pdf;
62     }
63
64     return 0.0f;
65 }
66
67 inline Colour DirectedAreaLight::Li(Vector& Wi, Intersection& intersect)
68     const
69 {
70     return 0.0f;
71 }
72
73 inline Colour DirectedAreaLight::Le(const Ray& ray, Intersection&
74     intersection) const
75 {
76     return PMaterial->F(intersection, ray.GetDirection(),
77     (intersection
78         .GetPoint() - ray.GetOrigin()));
79 }
80
81 inline Colour DirectedAreaLight::L(const Point& p, const Normal& n,
82     const Vector& wo, Intersection& intersection) const
83 {
84     bool IsOnEmittanceSide = Dot(n, wo) > 0.0f;
85     if (IsOnEmittanceSide)
86         return M_Radiance * PMaterial->F(intersection, wo,
87             (p -
88                 intersection.GetPoint()));
89     return Colour(0.0f);
90 }
91
92 inline bool DirectedAreaLight::IsArea() const
93 {
94     return true;
95 }
96
97
98

```

```
89 inline Colour DirectedAreaLight::GetPower() const
90 {
91     return M_Radiance * PGeometry->SurfaceArea() * M_PI;
92 }
93
94 inline Point DirectedAreaLight::GetPosition() const
95 {
96     return Point(0, 0, 0);
97 }
```

2.7.4 DistantDirectionalLight

Listing 2.64: DistantDirectionalLight header file

```
1 #pragma once
2
3 #include "../Light.h"
4 #include "../../Core/Math/Point2D.h"
5
6 class DistantDirectionalLight : public Light
7 {
8 public:
9
10    DistantDirectionalLight(Transform& localToWorld, const Colour&
11        intensity, Vector direction)
12        : Light(localToWorld),
13        Intensity(intensity),
14        Direction(LocalToWorld * direction)
15    {
16    }
17
18
19    Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
20        Intersection& intersection) const override;
21    Colour Li(Vector& Wi, Intersection& intersect) const override;
22    Point GetPosition() const override;
23    Colour GetPower() const override;
24
25    bool IsInfinite() const override;
26
27 private:
28    Colour Intensity;
29    Vector Direction;
30 };
31 inline Colour DistantDirectionalLight::SampleLight(const Point2D&
32     sample, Point& point, Vector& wi, Intersection& intersection) const
33 {
34     wi = Direction;
35     return Intensity;
36 }
37 inline Colour DistantDirectionalLight::Li(Vector& Wi, Intersection&
38     intersect) const
39 {
40     return 0.0f;
41 }
42 inline Point DistantDirectionalLight::GetPosition() const
43 {
44     return Point(0, 0, 0);
45 }
```

```
46
47 inline Colour DistantDirectionalLight::GetPower() const
48 {
49     return 0.0f;
50 }
51
52 inline bool DistantDirectionalLight::IsInfinite() const
53 {
54     return true;
55 }
```

2.7.5 DistantDiscLight

Listing 2.65: DistantDiscLight header file

```
1 #pragma once
2
3 #include "../Light.h"
4 #include "../Texture.h"
5 #include "../Mapping.h"
6 #include "AreaLight.h"
7 #include "../Sample.h"
8
9 class DistantDiscLight : public AreaLight
10 {
11 public:
12
13     DistantDiscLight(Transform& localToWorld, Colour& power, float
14         angle)
15         : AreaLight(localToWorld),
16             M_Radiance(power),
17             M_Theta(DegToRad(angle))
18     {}
19
20     Colour L(const Point& p, const Normal& n, const Vector& wo,
21             Intersection& intersection) const override;
22     virtual Colour Li(Vector& wi, Intersection& intersection) const
23             override
24     {
25         //Point2D uv = Mapping::Spherical(wi);
26         //Point2D uv = Mapping::Angular(wi);
27         //std::cout << DLTexture->GetValue(uv.GetX(), uv.GetY(),
28             intersection) << std::endl;
29         //return DLTexture->GetValue(uv.GetX(), uv.GetY(), intersection
30             );
31         return 0.0f;
32     }
33
34     Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
35                         Intersection& intersection) const override;
36
37     Colour Le(const Ray& ray, Intersection& intersection) const
38             override;
39     bool IsInfinite() const override;
40
41     Point GetPosition() const override;
42     Colour GetPower() const override;
43
44 private:
45     Texture* DLTexture;
46     Colour M_Radiance;
47     float M_Theta;
48 }
```

```

43 inline Colour DistantDiscLight::L(const Point& p, const Normal& n,
44     const Vector& wo, Intersection& intersection) const
45 {
46     return 0.0f;
47 }
48 inline Colour DistantDiscLight::SampleLight(const Point2D& sample,
49     Point& point, Vector& wi, Intersection& intersection) const
50 {
51     float cosThetaMax = (1.0f - cosf(M_Theta)) / 2.0f;
52     //Point2D sample = Point2D(rand() / float(RAND_MAX), rand() / float
53     //    (RAND_MAX));
54     wi = (LocalToWorld * Sample::UniformSphericalCap(Point2D(sample[0],
55         sample[1]), cosThetaMax)).Hat();
56     // pdf
57     float pdf = 1.0f / (4.0f * M_PI * cosThetaMax);
58     if (pdf <= 0.0f)
59     {
60         return Colour(0.0f);
61     }
62     return M_Radiance / pdf;
63 }
64 inline Colour DistantDiscLight::Le(const Ray& ray, Intersection&
65     intersection) const
66 {
67     float cosThetaMax = (1.0f - cosf(M_Theta)) / 2.0f;
68     Vector lightDir = (Vector(0, 1, 0)).Hat();
69     if (Dot(WorldToLocal * ray.GetDirection(), lightDir) < cosf(M_Theta
70         ))
71     {
72         return Colour(0.0f);
73     }
74     return M_Radiance;
75 }
76 inline bool DistantDiscLight::IsInfinite() const
77 {
78     return true;
79 }
80
81 inline Point DistantDiscLight::GetPosition() const
82 {
83     return Point(0, 0, 0);
84 }
85
86 inline Colour DistantDiscLight::GetPower() const
87 {
88     return 0.0f;

```


2.7.6 Dome

Listing 2.66: Dome header file

```
1 #pragma once
2
3 #include "../Light.h"
4 #include "../Texture.h"
5 #include "../Mapping.h"
6 #include "../Sample.h"
7 #include "AreaLight.h"
8 #include "../Textures/ToneMapping.h"
9 #include "../../Core/Math/Statistics.h"
10 #include "../Textures/BitmapTexture.h"
11
12 class DomeLight : public AreaLight
13 {
14 public:
15
16     DomeLight(Transform& localToWorld, Colour& power, Texture* texture,
17               float multiplier = 1.0)
18         : AreaLight(localToWorld),
19           DLTexture(texture),
20           DLMultiplier(multiplier),
21           DLSampler(((BitmapTexture*)texture)->GetBitmap())
22     {}
23
24     Colour L(const Point& p, const Normal& n, const Vector& wo,
25               Intersection& intersection) const override;
26     virtual Colour Li(Vector& wi, Intersection& intersection) const
27         override
28     {
29         //Point2D uv = Mapping::Spherical(wi);
30         //Point2D uv = Mapping::Angular(wi);
31         //std::cout << DLTexture->GetValue(uv.GetX(), uv.GetY(),
32         //    intersection) << std::endl;
33         //return DLTexture->GetValue(uv.GetX(), uv.GetY(), intersection
34         //    );
35         return 0.0f;
36     }
37
38     Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
39                         Intersection& intersection) const override;
40
41     Colour Le(const Ray& ray, Intersection& intersection) const
42         override;
43     bool IsInfinite() const override;
44
45     Point GetPosition() const override;
46     Colour GetPower() const override;
47
48 private:
49     Texture* DLTexture;
50     float DLMultiplier;
```

```

43     Distribution2D DLSampler;
44
45 }
46
47 inline Colour DomeLight::L(const Point& p, const Normal& n, const
48     Vector& wo, Intersection& intersection) const
49 {
50     return 0.0f;
51 }
52
53 inline Colour DomeLight::SampleLight(const Point2D& sample, Point&
54     point, Vector& wi, Intersection& intersection) const
55 {
56 #if 1
57     Point2D uv;// = Point2D(p.GetX(), p.GetY());
58     float pdf;
59     DLSampler.Sample2D(Point2D(sample.GetX(), sample.GetY()), uv, pdf);
60     float theta = uv[1] * M_PI;
61     float phi = uv[0] * 2 * M_PI;
62     float costheta = cosf(theta);
63     float sintheta = sinf(theta);
64     float cosphi = cosf(phi);
65     float sinphi = sinf(phi);
66     wi = (WorldToLocal * Vector(sintheta* cosphi, costheta, sintheta*
67         sinphi)).Hat();
68     point = Point(wi[0], wi[1], wi[2]) * 1e5;
69     Bitmap* bitmap = ((BitmapTexture*)DLTexture)->GetBitmap();
70     pdf = pdf * (bitmap->GetWidth() * bitmap->GetHeight()) / (2 * M_PI
71         * M_PI * sintheta);
72
73     if (pdf == 0.0f)
74     {
75         return 0.0f;
76     }
77
78     return (bitmap->GetPixel(uv[0] * bitmap->GetWidth(), uv[1] * bitmap
79         ->GetHeight())) / (pdf * M_PI) * DLMultiplier;
80 #else
81     wi = Sample::UniformSphere(Point2D(p[0], p[1]));
82 //wi = Vector(Sample::CosineHemisphere(Point2D(p[0], p[1])));
83 //float pdf = Sample::CosineHemispherePDF(wi[1]);
84     float pdf = Sample::UniformSpherePDF();
85     Point2D uv = Mapping::Spherical(wi);
86 //return (DLTexture->GetValue(uv[0], uv[1], intersection));
87     Bitmap* bitmap = ((BitmapTexture*)DLTexture)->GetBitmap();
88     return bitmap->GetPixel(uv[0] * (bitmap->GetWidth()-1), uv[1] *
89         bitmap->GetHeight()-1)) / pdf;
90 #endif
91 }
92
93
94 inline Colour DomeLight::Le(const Ray& ray, Intersection& intersection)
95     const

```

```
88  {
89      Vector localDirection(WorldToLocal * ray.GetDirection());
90      Point2D uv(Mapping::Spherical(localDirection.Hat()));
91      return DLTexture->GetValue(uv.GetX(), uv.GetY(), intersection) *
92          DLMultiplier;
93  }
94  inline bool DomeLight::IsInfinite() const
95  {
96      return true;
97  }
98
99  inline Point DomeLight::GetPosition() const
100 {
101     return Point(0,0,0);
102 }
103
104 inline Colour DomeLight::GetPower() const
105 {
106     return 0.0f;
107 }
```

2.7.7 IES

Listing 2.67: IES header file

```
1 #pragma once
2
3 #include <stdio.h>
4 #include <string>
5 #include <fstream>
6 #include <stdio.h>
7 #include <sstream>
8 #include <regex>
9 #include <algorithm>
10 #include <iostream>
11 /**
12 /**
13
14 struct IESKeyword
15 {
16     std::string test;
17     std::string manufacturer;
18     std::string luminaireCatalogNumber;
19     std::string luminaireDescription;
20     std::string lampCatalogNumber;
21     std::string lampDescription;
22
23 };
24
25 struct IESInformation
26 {
27     int numberofLamps;
28     float lumensPerLamp;
29     float candelamultiplier;
30     int numberofVerticalAngles;
31     int numberofHorizontalAngles;
32     int photometricType;
33     int unitType;
34     float width;
35     float length;
36     float height;
37 };
38
39 struct IESBallast
40 {
41     float ballastFactor;
42     float ballastLampPhotometricFactor;
43     float inputWatt;
44 };
45
46 struct IESData
47 {
48     IESInformation info;
49     IESBallast ballast;
```

```

50     std::vector<float> verticalAngles;
51     std::vector<float> horizontalAngles;
52     std::vector<float> verticalCandela;
53     std::vector<float> horizontalCandela;
54 }
55
56 std::ostream& operator<<(std::ostream& os, const IESInformation& i)
57 {
58     os << "numberOfLamps:"           " << i.numberOfLamps        <<
59     os << "lumensPerLamp:"         " << i.lumensPerLamp        <<
60     os << "candelaMultiplier;"    " << i.candelaMultiplier    <<
61     os << "numberOfVerticalAngles;" " << i.numberOfVerticalAngles <<
62     os << "numberOfHorizontalAngles;" " << i.numberOfHorizontalAngles <<
63     os << "photometricType;"      " << i.photometricType      <<
64     os << "unitType;"             " << i.unitType            <<
65     os << "width;"               " << i.width               <<
66     os << "length;"              " << i.length              <<
67     os << "height;"              " << i.height              <<
68     std::endl;
69 }
70 std::istream& operator>>(std::istream& is, IESInformation& info)
71 {
72     is >> info.numberOfLamps >> info.lumensPerLamp;
73     is >> info.candelaMultiplier;
74     is >> info.numberOfVerticalAngles >> info.numberOfHorizontalAngles;
75     is >> info.photometricType >> info.unitType;
76     is >> info.width >> info.length >> info.height;
77     return is;
78 }
79
80 std::istream& operator>>(std::istream& is, IESBallast& b)
81 {
82     is >> b.ballastFactor;
83     is >> b.ballastLampPhotometricFactor;
84     is >> b.inputWatt;
85     return is;
86 }
87
88 namespace SPEC
89 {
90     std::string LM_63_1995 = "IESNA:LM-63-1995";
91     std::string LM_63_1991 = "IESNA91";

```

```

92     std::string LM_63_1986 = "";
93 }
94
95 inline void LoadIES(std::string filename, IESData& iesData)
96 {
97     int lineNumber = 0;
98     std::ifstream InputFileStream(filename);
99     if (!InputFileStream)
100    {
101        std::cerr << "Failed to load file: " << filename << std::endl;
102    }
103    std::string Line;
104    while (!InputFileStream.eof())
105    {
106        std::getline(InputFileStream, Line);
107        if (Line == SPEC::LM_63_1995)
108        {
109        }
110        else if (Line[0] == '[')
111        {
112            std::string temp = "";
113            std::string value = "";
114            std::string keyword = "";
115            for (char& letter : Line)
116            {
117
118                if (letter == ']')
119                {
120                    keyword = temp;
121                    temp = "";
122                }
123                else if (letter != '[')
124                {
125                    temp += letter;
126                }
127            }
128            value = temp;
129            std::cout << keyword << " " << value << std::endl;
130            lineNumber++;
131        }
132        else if (std::size_t pos = Line.find("TILT=") != std::string::npos)
133        {
134            std::string tilt = "";
135            for (int i = 5; i != Line.size(); ++i)
136            {
137                tilt += Line[i];
138            }
139            std::cout << tilt << std::endl;
140            break;
141        }
142    }

```

```

143
144 }
145 IESInformation info;
146 InputFileStream >> info;
147 std::cout << info << std::endl;
148 IESBallast ballast;
149 InputFileStream >> ballast;
150 std::cout << " " << ballast.ballastFactor << " " << ballast.
    ballastLampPhotometricFactor << " " << ballast.inputWatt << std
    ::endl;

151
152 std::vector<float> verticalAngles;
153 float temp;
154 for (int i = 0; i < info.numberOfVerticalAngles; ++i)
155 {
156     InputFileStream >> temp;
157     verticalAngles.push_back(temp);
158 }

159
160 std::vector<float> horizontalAngles;
161 for (int i = 0; i < info.numberOfHorizontalAngles; ++i)
162 {
163     InputFileStream >> temp;
164     horizontalAngles.push_back(temp);
165 }

166
167 std::vector<float> candelaVerticalAngles;
168 for (int i = 0; i < info.numberOfVerticalAngles; ++i)
169 {
170     InputFileStream >> temp;
171     candelaVerticalAngles.push_back(temp);
172 }

173
174 for (int i = 0; i < info.numberOfVerticalAngles; ++i)
175 {
176     std::cout << verticalAngles[i] << " " << candelaVerticalAngles[
        i] << std::endl;
177 }

178
179
180 std::vector<float> candelaHorizontalAngles;
181 for (int i = 0; i < info.numberOfHorizontalAngles; ++i)
182 {
183     InputFileStream >> temp;
184     candelaVerticalAngles.push_back(temp);
185 }

186
187 // Cubic interpolate data
188 std::vector<float> interpolatedValues;
189 std::vector<float> interpolatedAngles;
190 int count = 100;
191 float t = 0;

```

```

192     for (int c = 0; c < (info.numberOfVerticalAngles + c) / 4; ++c)
193     {
194         for (int i = 0; i < count; ++i)
195         {
196             t = i / float(count);
197             float f = CubicInterp(t,
198                         candelierVerticalAngles[c * 3],
199                         candelierVerticalAngles[c * 3 + 1],
200                         candelierVerticalAngles[c * 3 + 2],
201                         candelierVerticalAngles[c * 3 + 3]);
202             float a = CubicInterp(t,
203                         verticalAngles[c * 3],
204                         verticalAngles[c * 3 + 1],
205                         verticalAngles[c * 3 + 2],
206                         verticalAngles[c * 3 + 3]);
207             interpolatedValues.push_back(f);
208             interpolatedAngles.push_back(a);
209         }
210     }
211
212     iesData.info = info;
213     iesData.ballast = ballast;
214     iesData.verticalAngles = interpolatedAngles;
215     iesData.horizontalAngles = horizontalAngles;
216     iesData.verticalCandela = interpolatedValues;
217     iesData.horizontalCandela = candelierHorizontalAngles;
218
219     iesData.info.numberOfVerticalAngles = interpolatedValues.size();
220     InputFileStream.close();
221 }
```

2.7.8 IESLight

Listing 2.68: IESLight header file

```
1 #pragma once
2 #include "IES.h"
3 #include "../Light.h"
4
5 class IESLight : public Light
6 {
7 public:
8     IESLight(Transform& localToWorld, Colour& power, IESData& iesData,
9             float multiplier = 1.0);
10    virtual Colour Li(Vector& wi, Intersection& intersect) const
11        override;
12    Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
13                        Intersection& intersection) const override;
14    Colour GetPower() const override;
15    Point GetPosition() const override;
16    bool IsDelta() const override;
17
18 private:
19     Point Position;
20     Colour Intensity;
21     IESData Data;
22     float Multiplier;
23 };
24
25 inline IESLight::IESLight(Transform& localToWorld, Colour& intensity,
26                           IESData& iesData, float multiplier)
27     : Light(localToWorld),
28      Position(localToWorld * Point(0, 0, 0)),
29      Intensity(intensity),
30      Data(iesData),
31      Multiplier(multiplier)
32 {
33 }
34
35 inline Colour IESLight::Li(Vector& wi, Intersection& intersect) const
36 {
37     return 0.0f;
38 }
39
40 inline Colour IESLight::SampleLight(const Point2D& sample, Point& point
41                                     , Vector& wi, Intersection& intersection) const
42 {
43     Point intersectionPoint = intersection.GetPoint();
44     point = GetPosition();
45     wi = (Position - intersectionPoint).Hat();
```

```

45     float attenuation = DistanceSquared(Position, intersectionPoint);
46
47     if (attenuation == 0.0f)
48     {
49         return Colour(0,0,0);
50     }
51
52     // calculate angle from ies data
53     float costheta = OrthonormalBasis::CosTheta(WorldToLocal * wi);
54     float theta = acosf(costheta);
55     // Returns an angle between [0, PI]
56     float angle = RadToDeg(theta);
57
58     if (angle >= Data.verticalAngles[Data.verticalAngles.size() - 1])
59     {
60         return Colour(0,0,0);
61     }
62
63     // Calculate the angle->index factor into the vertical candela
64     // array
65     float increment = Data.info.numberOfVerticalAngles / (Data.
66             verticalAngles[Data.verticalAngles.size() - 1] + 1);
67     // interpolate candela values between angles
68     float d1 = Data.verticalCandela[angle * increment];
69     //int a1 = angle;
70     //float d2 = Data.verticalCandela[(angle + 1) * increment];
71     //float candela = lerp(d1, d2, angle - a1);
72     float candela = d1;
73
74     return Multiplier * Intensity * candela / attenuation;
75 }
76
77 inline Colour IESLight::GetPower() const
78 {
79     return Data.info.lumensPerLamp;
80 }
81
82 inline Point IESLight::GetPosition() const
83 {
84     return Position;
85 }
86
87 inline bool IESLight::IsDelta() const
88 {
89     return true;
90 }

```

2.7.9 PointLight

Listing 2.69: PointLight header file

```
1 #pragma once
2 #include "../Light.h"
3 #include "../../Core/Math/Point2D.h"
4
5 class PointLight : public Light
6 {
7 public:
8     PointLight();
9     PointLight(Transform& localToWorld, Colour& power, float multiplier
10         = 1.0);
11    virtual Colour Li(Vector& wi, Intersection& intersect) const
12        override;
13    Colour SampleLight(const Scene* scene, Sampler* sampler, Ray& ray,
14        Normal& normal) const override;
15    Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
16        Intersection& intersection) const override;
17    Colour GetPower() const override;
18    Point GetPosition() const override;
19
20    bool IsDelta() const override;
21
22 };
```

Listing 2.70: PointLight source file

```
1 #include "PointLight.h"
2 #include "../../Core/Utility/Misc.h"
3 #include "../../Core/Math/Point2D.h"
4 #include "../Sampler.h"
5 #include "../Sample.h"
6
7 PointLight::PointLight()
8 {
9
10 }
11
12 PointLight::PointLight(Transform& lightToWorld , Colour& power , float
13     multiplier)
14     : Light(lightToWorld) , Power(power) , PLMultiplier(multiplier)
15 {
16     Position = LocalToWorld * Point(0.0f, 0.0f, 0.0f);
17 }
18
19 Colour PointLight::Li(Vector& Wo , Intersection& intersection) const
20 {
21     return Power * PLMultiplier;
22 }
23
24 Colour PointLight::SampleLight(const Scene* scene , Sampler* sampler ,
25     Ray& ray , Normal& normal) const
26 {
27     ray = Ray(Position , Sample::UniformSphere(sampler->Sample2D()));
28     float pdf = Sample::UniformSpherePDF();
29     return Power * PLMultiplier / pdf;
30 }
31
32 Colour PointLight::SampleLight(const Point2D& sample , Point& point ,
33     Vector& wi , Intersection& intersection) const
34 {
35     point = GetPosition();
36     wi = (Position - intersection.GetPoint()).Hat();
37     float attenuation = DistanceSquared(Position , intersection.GetPoint
38         ());
39     float pdf = 4 * M_PI;
40     return Power * PLMultiplier / attenuation / pdf;
41 }
42
43 Colour PointLight::GetPower() const
44 {
45     return 4 * M_PI * Power;
46 }
47 Point PointLight::GetPosition() const
48 {
49     return Position;
```

```
48 }
49
50 bool PointLight::IsDelta() const
51 {
52     return true;
53 }
```

2.7.10 SpotLight

Listing 2.71: SpotLight header file

```
1 #pragma once
2
3 #include "../Light.h"
4
5 class SpotLight : public Light
6 {
7 public:
8     SpotLight(Transform& localToWorld, const Colour& intensity, float
9             hotspot, float falloff, float tightness = 0.0f)
10    : Light(localToWorld),
11      Intensity(intensity),
12      Position(localToWorld * Point(0,0,0)),
13      Tightness(tightness)
14    {
15        InnerCone = cosf(DegToRad(hotspot));
16        OuterCone = cosf(DegToRad(falloff));
17    }
18
19
20    Colour SampleLight(const Point2D& sample, Point& point, Vector& wi,
21                      Intersection& intersection) const override;
22    Colour Li(Vector& Wi, Intersection& intersect) const override;
23    Point GetPosition() const override;
24    Colour GetPower() const override;
25
26    float CalculateFalloff(Vector& lightDir) const;
27
28 private:
29     Colour Intensity;
30     Point Position;
31     float InnerCone;
32     float OuterCone;
33     float Tightness;
34 };
35 inline Colour SpotLight::SampleLight(const Point2D& sample, Point&
36                                     point, Vector& wi, Intersection& intersection) const
37 {
38     point = GetPosition();
39     wi = (Position - intersection.GetPoint()).Hat();
40     // Calculate falloff
41     return Intensity * CalculateFalloff(wi) / DistanceSquared(
42         intersection.GetPoint(), Position);
43 }
44
45 inline Colour SpotLight::Li(Vector& Wi, Intersection& intersect) const
46 {
47     return 0.0f;
```

```

46 }
47
48 inline Point SpotLight::GetPosition() const
49 {
50     return Position;
51 }
52
53 // See RBRT 614
54 // See https://en.wikipedia.org/wiki/Solid_angle#Cone.2C_spherical_cap
55 // .2C_hemisphere
55 inline Colour SpotLight::GetPower() const
56 {
57     return Intensity * 2.0f * M_PI * (1.0f - 0.5f * (InnerCone -
58         OuterCone));
59 }
60
61 // See PBRT Chapter 12 pg.614
62 // See http://learnopengl.com/#!Lighting/Light-casters
63 // See http://www.povray.org/documentation/view/3.7.0/310/
63 inline float SpotLight::CalculateFalloff(Vector& wi) const
64 {
65     Vector localLightDir = (WorldToLocal * wi).Hat();
66     float costheta = localLightDir.GetY();
67     if (costheta < OuterCone)
68     {
69         return 0.0f;
70     }
71     if (costheta > InnerCone)
72     {
73         return 1.0f;
74     }
75
76     float intensity = (costheta - OuterCone) / (InnerCone - OuterCone);
77     return powf(intensity, Tightness);
78 }

```

2.8 Materials

2.8.1 BlendMaterial

Listing 2.72: BlendMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../../Core/Utility/Misc.h"
5 #include "../Texture.h"
6
7 class BlendMaterial : public Material
8 {
9 public:
10
11     BlendMaterial(Material* baseMaterial, Material* coatMaterial,
12                  Texture* blendMap = nullptr, float blendAmount = 1.0f)
13     : BaseMaterial(baseMaterial), CoatMaterial(coatMaterial),
14       BlendMap(blendMap), BlendAmount(blendAmount)
15     {}
16
17     // Evaluates the BRDF
18     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
19     const override
20     {
21         float amount = BlendAmount;
22         if (BlendMap != nullptr)
23         {
24             amount *= BlendMap->GetValue(intersection.U, intersection.V,
25                                             intersection)[0];
26         }
27         return (1 - amount) * BaseMaterial->F(intersection, Wo, Wi) +
28                amount * CoatMaterial->F(intersection, Wo, Wi);
29     }
30
31     Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
32                     float& Pdf) const override
33     {
34         float amount = BlendAmount;
35         if (BlendMap != nullptr)
36         {
37             amount *= BlendMap->GetValue(intersection.U, intersection.V,
38                                             intersection)[0];
39         }
40
41         float u = rand() / float(RAND_MAX);
42         if (u < amount)
43         {
44             return CoatMaterial->SampleF(intersection, Wo, Wi, Pdf);
45         }
46     }
47 }
```

```
40
41     return BaseMaterial->SampleF(intersection, Wo, Wi, Pdf);
42     return 0.0f;
43 }
44
45 BRDF::Type GetType() const override
46 {
47     return BaseMaterial->GetType() + CoatMaterial->GetType();
48 }
49
50 private:
51     Material* BaseMaterial;
52     Material* CoatMaterial;
53     Texture* BlendMap;
54     float BlendAmount;
55 }
```

2.8.2 BlinnMaterial

Listing 2.73: BlinnMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../Texture.h"
5 #include "../BRDF/BlinnMicrofacetBRDF.h"
6
7 class BlinnMaterial : public Material
8 {
9 public:
10
11     BlinnMaterial(float roughness, float ior, Texture* diffuse, Texture*
12                  * bump = nullptr)
13         : Roughness(roughness), DiffuseTexture(diffuse), BumpTexture(
14             bump)
15     {
16         blinn = new BlinnMicrofacetBRDF(roughness, diffuse, ior);
17     }
18
19     // Evaluates the BRDF
20     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
21         const override
22     {
23         if (BumpTexture != nullptr)
24         {
25             NormalBump(intersection, BumpTexture, true, false, 0.5f);
26         }
27         Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).
28             Hat();
29         Wi = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wi).
30             Hat();
31         return blinn->F(Wo, Wi, intersection);
32     }
33
34     Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
35                     float& Pdf) const override;
36
37     BRDF::Type GetType() const override
38     {
39         return blinn->GetType();
40     }
41
42 private:
43     Texture* DiffuseTexture;
44     Texture* BumpTexture;
45     float Roughness;
46     BlinnMicrofacetBRDF* blinn;
47 },
```

```
44 inline Colour BlinnMaterial::SampleF(Intersection& intersection, Vector
45   Wo, Vector& Wi, float& Pdf) const
46 {
47     Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).Hat()
48     ;
49     Colour F = blinn->SampleF(Wo, Wi, Pdf, intersection);
50     Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).Hat()
51     ;
52     return F;
53 }
```

2.8.3 DielectricMaterial

Listing 2.74: DielectricMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../Texture.h"
6 #include "../../Core/Utility/Misc.h"
7 #include "../BRDF/GlossySpecular.h"
8 #include "../BRDF/SpecularReflection.h"
9 #include "../BRDF/Conductor.h"
10 #include "../BRDF/Dielectric.h"
11 #include "../BRDF/DielectricBSDF.h"
12
13 class DielectricMaterial : public Material
14 {
15 public:
16     BRDF::Type GetType() const override;
17
18     DielectricMaterial(Colour kd, float ior)
19         : Kd(kd),
20           IOR(ior)
21     {
22         SpecularBRDF = new DielectricBSDF(Kd, 1.0, ior);
23         Dielec = new Dielectric(1, ior);
24     }
25
26     // Evaluates the BRDF
27     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
28         const override
29     {
30         return 0.0f;
31     }
32
33     Colour Rho(Vector& wo, Intersection& intersection) const override
34     {
35         return SpecularBRDF->Rho(wo, intersection);
36     }
37
38     virtual Colour SampleF(Intersection& intersection, Vector Wo,
39                           Vector& Wi, float& Pdf) const
40     {
41         //Normal n(intersection.GetNormal());
42         //float cos_thetai = Dot(n, Wo);
43         //float etai = 1.00;
44         //float etat = 1 / IOR;
45         //float eta = etai / etat;
46         //if (cos_thetai < 0.0f)
47         //{
48             //cos_thetai = -cos_thetai;
49             //n = -n;
```

```

48     //  eta = 1.0f / eta;
49     //}
50
51     //float temp = 1.0f - (1.0f - cos_thetai * cos_thetai) / (eta *
52     //  eta);
53     //if (temp < 0.0f)
54     //{
55     //  return (1.0f, 0.0f, 0.0f);
56     //}
57     //float cos_theta2 = sqrtf(temp);
58     //Wi = (-Wo / eta - (cos_theta2 - cos_thetai / eta) * Vector(n)
59     //  ).Hat();
60     //return ((eta * eta) *Colour(1, 1, 1) / fabsf(Dot(n, Wi)));
61
62     Colour Fr(0, 0, 0);
63     Vector w(intersection.GetNormal());
64     float pdf;
65     Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).
66         Hat();
67     Fr = SpecularBRDF->SampleF(Wo, Wi, pdf, intersection) * Kd;//
68     *(Colour(1) - Dielec->Evaluate(OrthonormalBasis::CosTheta(
69     Wo)));
70     Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).
71         Hat();
72     return Fr;
73
74 private:
75     Colour Kd; // Diffuse coefficient
76     DielectricBSDF* SpecularBRDF;
77     Dielectric* Dielec;
78     float IOR;
79 };
80
81 inline BRDF::Type DielectricMaterial::GetType() const
82 {
83     //return SpecularBRDF->GetType();
84     return BRDF::Type::REFRACTION;
85 }

```

2.8.4 DiffuseMaterial

Listing 2.75: DiffuseMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../Texture.h"
6 #include "../../Core/Utility/Misc.h"
7
8 class DiffuseMaterial : public Material
9 {
10 public:
11
12     DiffuseMaterial(Texture* diffuse, Colour kd, Texture* bumpTexture =
13                     nullptr, float bumpAmount = 1.0f)
14         : DiffuseTexture(diffuse), Kd(kd), BumpTexture(bumpTexture),
15           BumpAmount(bumpAmount)
16     {
17         DiffuseBRDF = (new Lambertian(diffuse));
18     }
19
20     // Evaluates the BRDF
21     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
22         const override
23     {
24         if (BumpTexture != nullptr)
25         {
26             NormalBump(intersection, BumpTexture, false, false,
27                         BumpAmount);
28         }
29         Colour Fr(0, 0, 0);
30         Fr = DiffuseBRDF->F(Wo, Wi, intersection) * M_PI;
31         return Fr;
32     }
33
34     Colour Rho(Vector& wo, Intersection& intersection) const override
35     {
36         return DiffuseBRDF->Rho(wo, intersection);
37     }
38
39     Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
40                      float& Pdf) const override;
41
42
43 private:
44     Colour Kd; // Diffuse coefficient
```

```
45     Texture* DiffuseTexture;
46     Lambertian* DiffuseBRDF;
47     Texture* BumpTexture;
48     float BumpAmount;
49 }
50
51 inline Colour DiffuseMaterial::SampleF(Intersection& intersection,
52                                         Vector Wo, Vector& Wi, float& Pdf) const
52 {
53     if (BumpTexture != nullptr)
54     {
55         NormalBump(intersection, BumpTexture, false, false, BumpAmount)
56         ;
57     }
58     Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo);
59     Colour F = DiffuseBRDF->SampleF(Wo, Wi, Pdf, intersection);
60     Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi);
61     return F;
61 }
```

2.8.5 EmissiveMaterial

Listing 2.76: EmissiveMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../Texture.h"
5
6 class EmissiveMaterial : public Material
7 {
8 public:
9
10    EmissiveMaterial(Texture* diffuse, float multiplier)
11        : DiffuseTexture(diffuse), Multiplier(multiplier)
12    {
13    }
14
15    virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
16        const override
17    {
18        return Multiplier * DiffuseTexture->GetValue(intersection.U,
19            intersection.V, intersection);
20    }
21
22    Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
23        float& Pdf) const override;
24
25    BRDF::Type GetType() const override
26    {
27        return BRDF::Type::EMISSIVE;
28    }
29
30 private:
31    Texture* DiffuseTexture;
32    float Multiplier;
33 };
34 inline Colour EmissiveMaterial::SampleF(Intersection& intersection,
35     Vector Wo, Vector& Wi, float& Pdf) const
36 {
37     Pdf = 0.0f;
38     return Multiplier * DiffuseTexture->GetValue(intersection.U,
39         intersection.V, intersection);
40 }
```

2.8.6 PhongMaterial

Listing 2.77: PhongMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../BRDF/GlossySpecular.h"
6 #include "../../Core/Utility/Misc.h"
7
8 #include "../Texture.h"
9
10 class PhongMaterial : public Material
11 {
12 public:
13     PhongMaterial(Colour kd, Colour ks, float specularLevel, float
14                     glossiness, Texture* diffuse, Texture* bump = nullptr)
15         : Kd(kd), Ks(ks), SpecularLevel(specularLevel), Glossiness(
16             glossiness), DiffuseTexture(diffuse), BumpTexture(bump)
17     {
18         //shader.AddBRDF(new Lambertian(Kd));
19         //shader.AddBRDF(new GlossySpecular(Ks));
20         DiffuseBRDF = new Lambertian(diffuse);
21         SpecularBRDF = new GlossySpecular(Ks);
22     }
23
24     // Evaluates the BRDF
25     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
26         const override
27     {
28         // The light back towards camera
29         Colour L(0, 0, 0);
30         if (BumpTexture != nullptr)
31         {
32             NormalBump(intersection, BumpTexture, true, false, 0.5f);
33         }
34         Normal normal = intersection.GetNormal();
35         float DotNWi = Dot(Wi, normal);
36         Wi = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wi).
37             Hat();
38         // Note that we must normalize here because we get silly
39         // artifacts otherwise
40         Vector ReflectionDirection = Reflect(Wi, normal).Hat();
41         // Why we multiply by PI
42         // https://seblagarde.wordpress.com/2012/01/08/pi-or-not-to-pi-
43         // in-game-lighting-equation/
44         L += DiffuseBRDF->F(Wo, Wi, intersection);
45         L += SpecularLevel * pow(Max(0.0f, Dot(Wo, ReflectionDirection)
46             ), pow(2, Glossiness / 10));
47         L = L * DiffuseTexture->GetValue(intersection.U, intersection.V
48             , intersection);
49     }
50     return L;
51 }
```

```
42     }
43
44     BRDF::Type GetType() const override
45     {
46         return DiffuseBRDF->GetType() + SpecularBRDF->GetType();
47     }
48
49 private:
50     Colour Kd; // Diffuse coefficient
51     Colour Ks; // Specular coefficient
52     Lambertian* DiffuseBRDF;
53     GlossySpecular* SpecularBRDF;
54     float SpecularLevel;
55     Texture* DiffuseTexture;
56     Texture* BumpTexture;
57     //Shader shader;
58     // mapping 0->1, 10->2, 20->4, 40->16
59     float Glossiness;
60 }
```

2.8.7 Plastic

Listing 2.78: Plastic header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../BRDF/GlossySpecular.h"
6 #include "../../Core/Utility/Misc.h"
7
8 #include "../Texture.h"
9 #include "../BRDF/BlinnMicrofacetBRDF.h"
10
11 class PlasticMaterial1 : public Material
12 {
13 public:
14
15     PlasticMaterial1(Colour kd, Colour ks, float roughness, float ior,
16                         Texture* diffuse = nullptr, Texture* specular = nullptr,
17                         Texture* reflection = nullptr, Texture* bump = nullptr, float
18                         bumpAmount = 1.0f)
19         : Kd(kd), Ks(ks), Roughness(roughness), IOR(ior),
20           DiffuseTexture(diffuse), ReflectionTexture(reflection),
21           BumpTexture(bump), BumpAmount(bumpAmount)
22     {
23         blinn = new BlinnMicrofacetBRDF(roughness, nullptr, ior,
24                                         specular);
25         lambertian = new Lambertian(diffuse);
26     }
27
28     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
29     const override
30     {
31         return lambertian->F(Wo, Wi, intersection) * M_PI * 2;
32     }
33
34
35 private:
36     Colour Kd; // Diffuse coefficient
37     Colour Ks; // Specular coefficient
38     float IOR;
39     Lambertian* lambertian;
40     BlinnMicrofacetBRDF* blinn;
41     float Roughness;
```

```

42     Texture* DiffuseTexture;
43     Texture* BumpTexture;
44     Texture* ReflectionTexture;
45     float BumpAmount;
46 };
47
48 inline Colour PlasticMaterial1::SampleF(Intersection& intersection,
49                                         Vector Wo, Vector& Wi, float& Pdf) const
50 {
51     if (BumpTexture != nullptr)
52     {
53         NormalBump(intersection, BumpTexture, false, false, BumpAmount);
54     }
55     Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).Hat();
56     //float pd = Kd[0] + Kd[1] + Kd[2];
57     //float ps = Ks[0] + Ks[1] + Ks[2];
58     //float u = (rand() / float(RAND_MAX)) * (pd + ps);
59     //
60     //Colour F;
61     //if (u < pd)
62     //{
63     //    Colour diffuse(Kd);
64     //    if (DiffuseTexture != nullptr)
65     //    {
66     //        diffuse = DiffuseTexture->GetValue(intersection.U,
67     //                                              intersection.V, intersection);
68     //    }
69     //    F = lambertian->SampleF(Wo, Wi, Pdf, intersection) * M_PI;
70     //} else
71     //{
72     //    Colour reflection(Ks);
73     //    if (ReflectionTexture != nullptr)
74     //    {
75     //        reflection = ReflectionTexture->GetValue(intersection.U,
76     //                                              intersection.V, intersection);
77     //    }
78     //    F = blinn->SampleF(Wo, Wi, Pdf, intersection) * reflection;
79     //}
80     Colour F;
81     Colour reflection(Ks);
82     if (ReflectionTexture != nullptr)
83     {
84         reflection = ReflectionTexture->GetValue(intersection.U,
85                                             intersection.V, intersection);
86     }
87     F = blinn->SampleF(Wo, Wi, Pdf, intersection) * reflection;
88     Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).Hat();
89     ;
90     return F;

```


2.8.8 RoughDielectricMaterial

Listing 2.79: RoughDielectricMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../Texture.h"
6 #include "../../Core/Utility/Misc.h"
7 #include "../../BRDF/GlossySpecular.h"
8 #include "../../BRDF/SpecularReflection.h"
9 #include "../../BRDF/Conductor.h"
10 #include "../../BRDF/Dielectric.h"
11 #include "../../BRDF/DielectricBSDF.h"
12 #include "DielectricMaterial.h"
13 #include "../../BRDF/RoughDielectricBRDF.h"
14
15 class RoughDielectricMaterial : public Material
16 {
17 public:
18     BRDF::Type GetType() const override;
19
20     RoughDielectricMaterial(Colour kd, float ior, float roughness,
21                             Texture* roughnessMap = nullptr)
22         : Kd(kd),
23           IOR(ior)
24     {
25         SpecularBRDF = new RoughDielectricBRDF(roughness, kd, ior, new
26                                               GGXDistribution(), roughnessMap);
27     }
28
29     // Evaluates the BRDF
30     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
31         const override
32     {
33         return 0.0f;
34     }
35
36     virtual Colour SampleF(Intersection& intersection, Vector Wo,
37                           Vector& Wi, float& Pdf) const
38     {
39         Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).
40              Hat();
41         Colour F = SpecularBRDF->SampleF(Wo, Wi, Pdf, intersection);
42         Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).
43              Hat();
44         return F;
45     }
46
47 private:
48     Colour Kd; // Diffuse coefficient
49     RoughDielectricBRDF* SpecularBRDF;
```

```
44     float IOR;
45 };
46
47 inline BRDF::Type RoughDielectricMaterial::GetType() const
48 {
49     return BRDF::Type::GLOSSY;
50 }
```

2.8.9 SpecularMaterial

Listing 2.80: SpecularMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../BRDF/Lambertian.h"
5 #include "../Texture.h"
6 #include "../../Core/Utility/Misc.h"
7 #include "../BRDF/GlossySpecular.h"
8 #include "../BRDF/SpecularReflection.h"
9 #include "../BRDF/Conductor.h"
10 #include "../BRDF/Dielectric.h"
11
12 class SpecularMaterial : public Material
13 {
14 public:
15     BRDF::Type GetType() const override;
16
17     SpecularMaterial(Colour kd, float ior, float k)
18         :Kd(kd)
19     {
20         SpecularBRDF = (new MirrorReflection(Kd, new Conductor(ior, k),
21                                             new RandomSampler(1)));
22     }
23
24     // Evaluates the BRDF
25     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
26         const override
27     {
28         return 0.0f;
29     }
30
31     virtual Colour SampleF(Intersection& intersection, Vector Wo,
32                            Vector& Wi, float& Pdf) const
33     {
34
35         Colour Fr(0, 0, 0);
36         Vector w(intersection.GetNormal());
37         float pdf;
38         Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo);
39         Fr = SpecularBRDF->SampleF(Wo, Wi, pdf, intersection) * Kd;
40         Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).
41             Hat();
42         return Fr * fabsf(Dot(Wi, w));
43     }
44
45 private:
46     Colour Kd; // Diffuse coefficient
47     MirrorReflection* SpecularBRDF;
48 };
49
```

```
46 inline BRDF::Type SpecularMaterial::GetType() const
47 {
48     return SpecularBRDF->GetType();
49 }
```

2.8.10 TwoSidedMaterial

Listing 2.81: TwoSidedMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../Texture.h"
5
6 class TwoSidedMaterial : public Material
7 {
8 public:
9
10    TwoSidedMaterial(Material* baseMaterial, Material* coatMaterial,
11                      Texture* blendMap = nullptr, float blendAmount = 0.0f, bool
12                      isTransluscent = false, bool flipDirection = false)
13        : FrontFacingMaterial(baseMaterial), BackFacingMaterial(
14            coatMaterial), BlendMap(blendMap), BlendAmount(blendAmount)
15            , Transluscent(isTransluscent), IncidentDirection(
16                flipDirection)
17    {
18    }
19
20    virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
21        const override
22    {
23        Vector direction = IncidentDirection ? Wi : Wo;
24        Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo);
25        Colour F;
26        if (Dot(intersection.GetNormal(), direction) < 0.0f)
27        {
28            F = BackFacingMaterial->F(intersection, Wo, Wi);
29            if (BlendMap)
30            {
31                F = F * BlendMap->GetValue(intersection.U, intersection
32                                              .V, intersection);
33            }
34
35            Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis,
36                                           Wi);
37            return F;
38        }
39
40        F = FrontFacingMaterial->F(intersection, Wo, Wi);
41        Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi);
42        return F;
43    }
44
45    Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
46                    float& Pdf) const override
47    {
48        Vector direction = IncidentDirection ? Wi : Wo;
49        Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo);
```

```

41
42     Colour F = FrontFacingMaterial->SampleF(intersection, Wo, Wi,
43                                         Pdf);
43     if (Dot(intersection.GetNormal(), direction) < 0.0f)
44     {
45         F = BackFacingMaterial->SampleF(intersection, Wo, Wi, Pdf);
46         if (BlendMap)
47         {
48             F = F * BlendMap->GetValue(intersection.U, intersection
49                                         .V, intersection);
50         }
51     }
52     Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi);
53     return F;
54 }
55
56
57 BRDF::Type GetType() const override
58 {
59     return FrontFacingMaterial->GetType() + BackFacingMaterial->
60           GetType();
61 }
62
63 bool IsTranslucent() const override
64 {
65     return Transluscent;
66 }
67 private:
68     Material* FrontFacingMaterial;
69     Material* BackFacingMaterial;
70     Texture* BlendMap;
71     float BlendAmount;
72     bool Transluscent;
73     bool IncidentDirection;
74 };

```

2.8.11 WardMaterial

Listing 2.82: WardMaterial header file

```
1 #pragma once
2
3 #include "../Material.h"
4 #include "../Texture.h"
5 #include "../BRDF/WardBRDF.h"
6
7 class WardMaterial : public Material
8 {
9 public:
10
11     WardMaterial(Colour ks, float ax, float ay, Texture* anisotropyMap
12                 = nullptr)
13     {
14         Ward = new WardBRDF(ks, ax, ay, anisotropyMap);
15     }
16
17     // Evaluates the BRDF
18     virtual Colour F(Intersection& intersection, Vector Wo, Vector Wi)
19             const override
20     {
21         return 0.0f;
22     }
23
24     Colour SampleF(Intersection& intersection, Vector Wo, Vector& Wi,
25                      float& Pdf) const override;
26
27     BRDF::Type GetType() const override
28     {
29         return BRDF::Type::GLOSSY;
30     }
31
32 private:
33     WardBRDF* Ward;
34 };
35
36 inline Colour WardMaterial::SampleF(Intersection& intersection, Vector
37                                     Wo, Vector& Wi, float& Pdf) const
38 {
39     //Wo = OrthonormalBasis::ToLocal(intersection.ShadingBasis, Wo).Hat
40     //();
41
42     OrthonormalBasis basis;
43     Vector w = Vector(intersection.IGNormal);
44     basis.W = Normalize(w);
45     if (fabs(w[1]) < 0.995f)
46         basis.U = (Vector(w[2], 0.0, -w[0]));
47     else
48         basis.U = Vector(0, w[2], -w[1]);
```

```
45 basis.V = Cross(w, basis.U).Hat();
46
47 Wo = OrthonormalBasis::ToLocal(basis, Wo).Hat();
48 Colour F = Ward->SampleF(Wo, Wi, Pdf, intersection);
49 Wi = OrthonormalBasis::ToWorld(basis, Wi).Hat();
50
51 //Wi = OrthonormalBasis::ToWorld(intersection.ShadingBasis, Wi).Hat
52   ();
53 return F;
54 }
```

2.9 Primitives

2.9.1 GeometricPrimitive

Listing 2.83: GeometricPrimitive header file

```
1 #pragma once
2
3 #include "../Primitive.h"
4 #include "../Geometry.h"
5
6 class GeometricPrimitive : public Primitive
7 {
8 public:
9     Material* GetMaterial() const override;
10    Geometry* GetGeometry() const;
11    std::vector<Primitive*> GetPrimitive() const override;
12    BBox GetBounds() const override;
13    bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
14                      Intersection& intersection) const override;
15    GeometricPrimitive(Geometry* geometry, Material* material);
16    bool IsCompound() const override;
17
18    virtual Point Sample(const Point2D& sample, Normal& normal) const
19                      override;
20
21 protected:
22     Geometry* PGeometry;
23     Material* PMaterial;
24 };
25
26 inline Material* GeometricPrimitive::GetMaterial() const
27 {
28     return PMaterial;
29 }
30
31 inline Geometry* GeometricPrimitive::GetGeometry() const
32 {
33     return PGeometry;
34 }
35
36 inline std::vector<Primitive*> GeometricPrimitive::GetPrimitive() const
37 {
38     std::vector<Primitive*> primitives;
39     if (PGeometry->IsCompound())
40     {
41         for (int i = 0; i < PGeometry->GetNumberOfGeometries(); ++i)
42         {
43             primitives.push_back(new GeometricPrimitive(PGeometry->
44                                     GetSubGeometry(i), PMaterial));
45         }
46     }
47 }
```

```

44     return primitives;
45 }
46
47 inline BBox GeometricPrimitive::GetBounds() const
48 {
49     return PGeometry->GetBounds();
50 }
51
52 inline bool GeometricPrimitive::IntersectRay(const Ray& ray, float&
53 tMin, float& epsilon, Intersection& intersection) const
54 {
55     if(PGeometry->IntersectRay(ray, tMin, epsilon, intersection))
56     {
57         intersection.material = PMaterial;
58         return true;
59     }
60     return false;
61 }
62
63 inline GeometricPrimitive::GeometricPrimitive(Geometry* geometry,
64 Material* material)
65 : PGeometry(geometry), PMaterial(material)
66 {
67
68 inline bool GeometricPrimitive::IsCompound() const
69 {
70     return PGeometry->IsCompound();
71 }
72 inline Point GeometricPrimitive::Sample(const Point2D& sample, Normal&
73 normal) const
74 {
75     return PGeometry->Sample(sample, normal);
76 }
```

2.9.2 InstancePrimitive

Listing 2.84: InstancePrimitive header file

```
1 #pragma once
2
3 #include "../Primitive.h"
4 #include "../../Core/Math/Transform.h"
5 class InstancePrimitive : public Primitive
6 {
7 public:
8     BBox GetBounds() const override;
9     bool IntersectRay(const Ray& ray, float& tMin, float& epsilon,
10                     Intersection& intersection) const override;
11    InstancePrimitive(Primitive* primitive, Material* material,
12                      Transform* transform);
13    Material* GetMaterial() const override;
14
15 private:
16     Transform* InstanceTransform;
17     Primitive* PrimitiveRef;
18     Material* PMaterial;
19 };
20
21 inline BBox InstancePrimitive::GetBounds() const
22 {
23     return (*InstanceTransform).operator*(PrimitiveRef->GetBounds());
24 }
25
26 inline bool InstancePrimitive::IntersectRay(const Ray& ray, float& tMin,
27                                              float& epsilon, Intersection& intersection) const
28 {
29     Ray localRay = InstanceTransform->Inverse() * ray;
30     if (!PrimitiveRef->IntersectRay(localRay, tMin, epsilon,
31                                     intersection))
32     {
33         return false;
34     }
35     intersection.INormal = (*InstanceTransform * intersection.INormal).
36     Hat();
37     intersection.IPoint = *InstanceTransform * intersection.IPoint;
38     intersection.ShadingBasis = OrthonormalBasis::FromW(Vector(
39         intersection.INormal));
40     intersection.material = PMaterial;
41     return true;
42 }
43
44 inline InstancePrimitive::InstancePrimitive(Primitive* primitive,
45                                             Material* material, Transform* transform)
46 : PrimitiveRef(primitive), PMaterial(material), InstanceTransform(
47     transform)
48 {
49 }
```

```
42
43 inline Material* InstancePrimitive::GetMaterial() const
44 {
45     return PMaterial;
46 }
```

2.10 Renderers

2.10.1 FunctionRenderer

Listing 2.85: FunctionRenderer header file

```
1 #pragma once
2
3 #include "../Renderer.h"
4 #include "../Camera.h"
5 #include "../Bitmap.h"
6
7 class FunctionRenderer : public Renderer
8 {
9 public:
10     FunctionRenderer(Scene* scene, Camera* camera, Integrator*
11                     integrator, Bitmap* bitmap, Sampler* sampler)
12         : scene(scene), camera(camera), integrator(integrator), bitmap(
13             bitmap), sampler(sampler)
14     {}
15
16     virtual void Render() const override
17     {
18         float CompletePercent = 0;
19         int Width = bitmap->GetWidth();
20         int Height = bitmap->GetHeight();
21 #pragma omp for ordered
22         for (int row = 0; row < Height; ++row)
23         {
24 #pragma omp parallel for
25             for (int col = 0; col < Width; ++col)
26             {
27                 Colour colour;
28                 for (int i = 0; i < sampler->GetNumberOfSamples(); ++i)
29                 {
30                     Point2D ap = sampler->Sample2D();
31                     float NDCx = (col + ap.GetX()) / Width;
32                     float NDCy = (row + ap.GetY()) / Height;
33                     // ndc->screen transformation
34                     float ScreenSpaceX = (2 * NDCx - 1);
35                     float ScreenSpaceY = (2 * NDCy - 1);
36                     Intersection intersection;
37                     //Ray ray = Ray(Point(0,0,0),Vector(NDCx, 1 - NDCy,
38                     //0));
39                     Ray ray = camera->CalculateRay(ScreenSpaceX,
40                         ScreenSpaceY, 0, 0, Width, Height);
41                     //Colour colour = render(col, row, Width, Height,
42                     //camera, sampler, geom, checkerMaskTexture, a,
43                     //light) ^ (1 / 2.2);
44                     colour = colour + Li(ray, sampler, intersection);
45                 }
46             }
47         }
48     }
49 }
```

```

41         bitmap->SetPixel(col, row, (colour * (1 / float(sampler
42             ->GetNumberOfSamples()) ^ (1 / 2.2)).Clamp()));
43         CompletePercent += 1 / float(Width);
44     }
45 //     if (func != nullptr)
46 //         func(0, row);
47     std::cout << "\r" << int(CompletePercent * 100.0 / Height)
48         << "% Rendering";
49 }
50
51 Colour Li(const Ray& ray, Sampler* sampler, Intersection& intersect
52 ) const
53 {
54     Colour li(0, 0, 0);
55     float tCurr = INFINITY;
56
57     li = integrator->Li(scene, ray, sampler);
58
59     return li;
60 }
61
62 Bitmap* bitmap;
63 private:
64     Scene* scene;
65     Camera* camera;
66     Integrator* integrator;
67     Sampler* sampler;
68 };

```

2.10.2 SimpleRenderer

Listing 2.86: SimpleRenderer header file

```
1 #pragma once
2
3 #include "../Renderer.h"
4 #include "../Camera.h"
5 #include "../Bitmap.h"
6
7 class SimpleRenderer : public Renderer
8 {
9 public:
10     SimpleRenderer(Scene* scene, Camera* camera, Integrator* integrator
11                 , Bitmap* bitmap, Sampler* sampler)
12         : scene(scene), camera(camera), integrator(integrator), bitmap(
13             bitmap), sampler(sampler)
14     {}
15
16     virtual void Render() const override
17     {
18         float CompletePercent = 0;
19         int Width = bitmap->GetWidth();
20         int Height = bitmap->GetHeight();
21         float InvSamples = 1 / float(sampler->GetNumberOfSamples());
22         float InvWidth = 1 / float(Width);
23         float InvHeight = 1 / float(Height);
24         float InvGamma = 1 / 2.2f;
25
26         #pragma omp parallel for
27         for (int row = 0; row < Height; ++row)
28         {
29             #pragma omp parallel for
30             for (int col = 0; col < Width; ++col)
31             {
32                 Colour colour;
33
34                 #pragma omp parallel for
35                 for (int i = 0; i < sampler->GetNumberOfSamples(); ++i)
36                 {
37                     Point2D ap = sampler->Sample2D();
38                     float SampleX = (col + ap.GetX());
39                     float SampleY = (row + ap.GetY());
40                     Intersection intersection;
41                     Ray ray = camera->CalculateRay(SampleX, SampleY, 0,
42                         0, Width, Height);
43                     colour += Li(ray, sampler, intersection) ;
44
45                 }
46                 //assert(colour.GetB() >= 0.0 && colour.GetG() >= 0.0
47                 //    && colour.GetR() >= 0.0);
48                 colour *= InvSamples;
```

```

45         bitmap->SetPixel(col, row, (colour ^ InvGamma).Clamp())
46             ;
47     }
48     if (func != nullptr)
49         func(0, row, bitmap->GetWidth(), 1, bitmap);
50 //std::cout << "\r" << int(CompletePercent * 100.0 / Height
51     ) << "% Rendering";
52 }
53 }
54
55 Colour Li(const Ray& ray, Sampler* sampler, Intersection& intersect
56 ) const
57 {
58     Colour li(0, 0, 0);
59     float tCurr = INFINITY;
60     if (scene->Intersect(ray, intersect))
61     {
62         li = integrator->Li(scene, ray, sampler);
63     }
64     else
65     {
66         return Colour(0.5, 0.5, 0.5);
67     }
68     return li;
69 }
70
71 Bitmap* GetBitmap()
72 {
73     return bitmap;
74 }
75
76
77     Bitmap* bitmap;
78 private:
79     Scene* scene;
80     Camera* camera;
81     Integrator* integrator;
82     Sampler* sampler;
83 };

```

2.10.3 TileRenderer

Listing 2.87: TileRenderer header file

```
1 #pragma once
2
3 #include "../Renderer.h"
4 #include "../Camera.h"
5 #include "../Bitmap.h"
6
7 class TileRenderer : public Renderer
8 {
9 public:
10     TileRenderer(Scene* scene, Camera* camera, Integrator* integrator,
11                  Bitmap* bitmap, Sampler* sampler, int tileSize)
12         : scene(scene), camera(camera), integrator(integrator), bitmap(
13             bitmap), sampler(sampler), tileSize(tileSize)
14     {
15         // round to the nearest tileSize
16         tilesx = (bitmap->GetWidth() + tileSize) / tileSize;
17         tilesy = (bitmap->GetHeight() + tileSize) / tileSize;
18         std::cout << tilesy << "," << tilesx << std::endl;
19         tiles = tilesx * tilesy;
20         Width = bitmap->GetWidth();
21         Height = bitmap->GetHeight();
22         shouldStopRendering = false;
23     }
24
25     virtual void Render() const override
26     {
27         //##pragma omp parallel for schedule(dynamic, 1)
28         //for (int currentTile = 0; currentTile < tiles; ++currentTile)
29         //{
30             // int row = currentTile / tilesx;
31             // int col = currentTile % tilesx;
32             // RenderTile(col, row);
33             //}
34             shouldStopRendering = false;
35             tilesx = (bitmap->GetWidth() + tileSize) / tileSize;
36             tilesy = (bitmap->GetHeight() + tileSize) / tileSize;
37             DIRECTION CurrentDirection = DIRECTION::DOWN;
38             int StepSquareSize = 0;
39             int StepsInCurrentSquare = -1;
40             int StepsInCurrentDirection = -1;
41             int Row = tilesy / 2;
42             int Column = tilesx / 2;
43         #pragma omp parallel for schedule(dynamic, 1)
44             for (int i = 0; i < Max(tilesx, tilesy) * Max(tilesx, tilesy);
45                  ++i)
46             {
47                 StepsInCurrentSquare++;
48             }
49         }
50     }
```

```

47     StepsInCurrentDirection++;
48     if (StepsInCurrentDirection == StepSquareSize)
49     {
50         StepsInCurrentDirection = 0;
51         switch (CurrentDirection)
52         {
53             case RIGHT:
54                 CurrentDirection = DOWN;
55                 break;
56             case DOWN:
57                 CurrentDirection = LEFT;
58                 break;
59             case LEFT:
60                 CurrentDirection = UP;
61                 break;
62             case UP:
63                 CurrentDirection = RIGHT;
64                 break;
65         }
66     }
67     if (StepsInCurrentSquare == (StepSquareSize + 1) * (
68         StepSquareSize + 1) - 1)
69     {
70         StepSquareSize++;
71     }
72     switch (CurrentDirection)
73     {
74         case RIGHT:
75             Column += 1;
76             break;
77         case DOWN:
78             Row += 1;
79             break;
80         case LEFT:
81             Column -= 1;
82             break;
83         case UP:
84             Row -= 1;
85             break;
86     }
87     if (!(Column < 0 || Column > tilesx) && !(Row < 0 || Row >
88         tilesy))
89     {
90         RenderTile(Column, Row);
91     }
92 }
93 }
94
95 void RenderTile(int col, int row) const
96 {

```

```

97     int x0 = col * tileSize;
98     int y0 = row * tileSize;
99     int x1 = Min(tileSize, bitmap->GetWidth() - x0);
100    int y1 = Min(tileSize, bitmap->GetHeight() - y0);
101    float InvGamma = 1 / 2.2f;
102
103    if(start_func != nullptr)
104        start_func(x0, y0, x1, y1);
105
106    for (int y = 0; y < y1; ++y)
107    {
108        if (shouldStopRendering)
109        {
110            tilesx = -1;
111            tilesy = -1;
112            // set previous work to black
113            for (int j = 0; j < y1; ++j)
114                for (int i = 0; i < x1; ++i)
115                    bitmap->SetPixel(x0 + i, y0 + j, Colour(0.0f));
116            if (func != nullptr)
117                func(x0, y0, x1, y1, bitmap);
118            return;
119        }
120        for (int x = 0; x < x1; ++x)
121        {
122            Colour colour;
123            for (int s = 0; s < sampler->GetNumberOfSamples(); ++s)
124            {
125
126                // Sample on image
127                Point2D ap = sampler->Sample2D();
128                Point2D lens = sampler->Sample2D();
129                float SampleX = (x0 + x + ap.GetX());
130                float SampleY = (y0 + y + ap.GetY());
131
132                Ray ray = camera->CalculateRay(SampleX, SampleY,
133                                                lens.GetX(), lens.GetY(), Width, Height);
134                Colour li = integrator->Li(scene, ray, sampler);
135                if (scene->GetVolume() != nullptr && scene->
136                    SVolumeIntegrator != nullptr)
137                {
138                    auto result = scene->SVolumeIntegrator->Li(
139                        scene, ray, sampler);
140                    colour += li * result.Transmittance + result.
141                        Luminance;
142                }
143            }
144            colour /= sampler->GetNumberOfSamples();

```

```

145         colour = (colour ^ InvGamma).Clamp();
146         bitmap->SetPixel(x0 + x, y0 + y, colour);
147     }
148
149     if (func != nullptr)
150         func(x0, y0, x1, y1, bitmap);
151 }
152
153 Colour Li(const Ray& ray, Sampler* sampler, Intersection& intersect
154             ) const
155 {
156     //Colour li(0, 0, 0);
157     //float tCurr = INFINITY;
158     //if (scene->Intersect(ray, intersect))
159     //{
160     //    //li = integrator->Li(scene, ray, sampler, intersect);
161     //    return integrator->Li(scene, ray, sampler);
162     //}
163     //else
164     //{
165         return Colour(0.5, 0.5, 0.5);
166     //}
167
168     //return li;
169 }
170
171 Bitmap* GetBitmap()
172 {
173     return bitmap;
174 }
175
176 void OnStop() override
177 {
178     shouldStopRendering = true;
179 }
180
181 Bitmap* bitmap;
182 mutable int tilesx;
183 mutable int tilesy;
184 int tiles;
185 int tileSize;
186 int Width;
187 int Height;
188 Random::RandomDouble rng;
189 enum DIRECTION
190 {
191     RIGHT,
192     DOWN,
193     LEFT,
194     UP
195 };

```

```
196 private:
197     Scene* scene;
198     Camera* camera;
199     Integrator* integrator;
200     Sampler* sampler;
201     mutable bool shouldStopRendering;
202 }
```

2.11 Sampler

2.11.1 RandomSampler

Listing 2.88: RandomSampler header file

```
1 #pragma once
2
3 #include <fstream>
4 #include <sstream>
5 #include "../Core/Utility/RNG.h"
6 #include "../Core/Math/Point2D.h"
7 #include "../Sampler.h"
8 #include "../Sample.h"
9
10 class RandomSampler : public Sampler
11 {
12 public:
13
14     RandomSampler(int samples)
15         : Sampler(samples), Index(0),
16           r(0.0, 1.0, 4245123)
17     {}
18
19     // Generates samples in a unit square
20     virtual void GenerateSamples() override
21     {
22         Random::RandomDouble r(0.0, 1.0, 4245123);
23         std::vector<Vector> hem;
24         for (int i = 0; i < NumberOfSamples; ++i)
25         {
26
27             Samples.push_back(Point2D(r(), r()));
28             //Sample::DiscConcentricSample(Samples[i], 1);
29             Sample::DiscRejectionSample(r, Samples[i]);
30         }
31     }
32
33     virtual float Sample1D() const override
34     {
35         return r();
36     }
37     // Get next 2D sample in unit square
38     virtual Point2D Sample2D() const override
39     {
40         //return Samples[Index++ % NumberOfSamples];
41         return Point2D(r(), r());
42     }
43
44     void Output(std::string path) const
45     {
46         std::ofstream ofs(path);
```

```

47     std::stringstream XCoordinates;
48     std::stringstream YCoordinates;
49
50     ofs << "clf;" << std::endl;
51     ofs << "graphics_toolkit(\"gnuplot\")" << std::endl;
52
53     XCoordinates << "x = [";
54     YCoordinates << "y = [";
55
56     if (NumberOfSamples > 1)
57     {
58         Point2D s = Samples[0];
59         XCoordinates << s.GetX();
60         YCoordinates << s.GetY();
61     }
62
63     for (int i = 1; i < NumberOfSamples; ++i)
64     {
65         Point2D s = Samples[i];
66         XCoordinates << ", " << s.GetX();
67         YCoordinates << ", " << s.GetY();
68     }
69     XCoordinates << "];" << std::endl;
70     YCoordinates << "];" << std::endl;
71
72     ofs << XCoordinates.str() << std::endl;
73     ofs << YCoordinates.str() << std::endl;
74
75     ofs << "c = x .* y;" << std::endl;
76     ofs << "scatter (x, y, 8, c, 'filled');" << std::endl;
77     ofs << "title ('Random Sampler with " << NumberOfSamples << "
78         samples');" << std::endl;
79     ofs << "grid on;" << std::endl;
80     // Sets major axis ticks to have correct spacing
81     float spacing = sqrtf(NumberOfSamples) / NumberOfSamples;
82     ofs << "xbounds = xlim();" << std::endl;
83     ofs << "set(gca, 'xtick', xbounds(1):" << spacing << ":xbounds
84         (2));" << std::endl;
85     ofs << "ybounds = ylim();" << std::endl;
86     ofs << "set(gca, 'ytick', ybounds(1):" << spacing << ":ybounds
87         (2));" << std::endl;
88     ofs << "set(gcf, 'PaperUnits', 'inches', 'PaperPosition', [0 0
89         4 4])" << std::endl;
90     ofs << "print -dpng F:\\temp\\plot.png";
91     ofs.close();
92 }
93
94 std::vector<Point2D> Samples;
95 Random::RandomDouble r;
96 private:
97     int Index;
98 };

```

2.11.2 StratifiedSampler

Listing 2.89: StratifiedSampler header file

```
1 #pragma once
2
3 #include <fstream>
4 #include <sstream>
5 #include "../../Core/Utility/RNG.h"
6 #include "../../Core/Math/Point2D.h"
7 #include "../Sampler.h"
8 #include "../Sample.h"
9
10 // Assumes that the number of samples is a perfect square
11 class StratifiedSampler : public Sampler
12 {
13 public:
14     StratifiedSampler(int samples, bool jitter)
15         : Sampler(samples), Index(0), IsJittered(jitter)
16     {}
17
18     // Generates samples in a unit square
19     virtual void GenerateSamples() override
20     {
21         Random::RandomDouble r(0.0, 1.0, 4245123);
22         int SqrtSamples = sqrtf(Number0fSamples);
23         for (int s = 0; s < Number0fSamples; ++s)
24         {
25             for (int i = 0; i < SqrtSamples; ++i)
26             {
27                 for (int j = 0; j < SqrtSamples; ++j)
28                 {
29                     if (IsJittered)
30                     {
31                         float x = (j + r()) / SqrtSamples;
32                         float y = (i + r()) / SqrtSamples;
33                         Samples.push_back(Point2D(x, y));
34                     }
35                     else
36                     {
37                         float x = (j + 0.5) / SqrtSamples;
38                         float y = (i + 0.5) / SqrtSamples;
39                         Samples.push_back(Point2D(x, y));
40                     }
41                 }
42             }
43         }
44     }
45 }
46
47     virtual float Sample1D() const override
48 {
49 }
```

```

50         return Samples[Index++ % Samples.size()][Index % 2];
51     }
52
53     // Get next 2D sample in unit square
54     virtual Point2D Sample2D() const override
55     {
56         return Samples[Index++ % Samples.size()];
57     }
58
59     void Shuffle()
60     {
61         random_shuffle(Samples.begin(), Samples.end());
62     }
63
64     void Output(std::string path) const
65     {
66         std::ofstream ofs(path);
67         std::stringstream XCoordinates;
68         std::stringstream YCoordinates;
69
70         ofs << "clf;" << std::endl;
71         // If you have trouble saving grid lines
72         ofs << "graphics_toolkit(\"gnuplot\");" << std::endl;
73         XCoordinates << "x = [";
74         YCoordinates << "y = [";
75
76         if (NumberOfSamples > 1)
77         {
78             Point2D s = Samples[0];
79             XCoordinates << s.GetX();
80             YCoordinates << s.GetY();
81         }
82
83         for (int i = 1; i < NumberOfSamples; ++i)
84         {
85             Point2D s = Samples[i];
86             XCoordinates << ", " << s.GetX();
87             YCoordinates << ", " << s.GetY();
88         }
89         XCoordinates << "];";
90         YCoordinates << "];";
91
92         ofs << XCoordinates.str() << std::endl;
93         ofs << YCoordinates.str() << std::endl;
94
95         ofs << "c = x .* y;" << std::endl;
96         ofs << "scatter (x, y, 8, c, 'filled');" << std::endl;
97         ofs << "title ('Stratified Sampler with " << NumberOfSamples <<
98             " samples, jittered=" << IsJittered << '));" << std::endl;
99         ofs << "grid on;" << std::endl;
100        // Sets major axis ticks to have correct spacing
101        float spacing = sqrtf(NumberOfSamples) / NumberOfSamples;

```

```
101     ofs << "xbounds = xlim%;" << std::endl;
102     ofs << "set(gca, 'xtick', xbounds(1):" << spacing << ":xbounds
103         (2)); " << std::endl;
104     ofs << "ybounds = ylim%;" << std::endl;
105     ofs << "set(gca, 'ytick', ybounds(1):" << spacing << ":ybounds
106         (2)); " << std::endl;
107     ofs << "set(gcf, 'PaperUnits', 'inches', 'PaperPosition', [0 0
108         4 4])" << std::endl;
109     ofs << "print -dpng F:\\temp\\plot.png";
110     ofs.close();
111 }
112
113 private:
114     std::vector<Point2D> Samples;
115     mutable int Index;
116     bool IsJittered;
```

2.12 Textures

2.12.1 BitmapTexture

Listing 2.90: BitmapTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../Bitmap.h"
4 #include <string>
5
6 class BitmapTexture : public Texture
7 {
8 public:
9     BitmapTexture() {}
10    BitmapTexture(Bitmap bitmap, float tileU = 1.0f, float tileV = 1.0f
11                  , float offsetU = 0.0f, float offsetV = 0.0f, bool filter =
12                  true, float gamma = 0.45454545);
13    BitmapTexture(std::string filename, float tileU = 1.0f, float tileV
14                  = 1.0f, float offsetU = 0.0f, float offsetV = 0.0f, bool
15                  filter = true, float gamma = 0.45454545);
16    BitmapTexture(std::string filename, std::string type, float tileU =
17                  1.0f, float tileV = 1.0f, float offsetU = 0.0f, float offsetV
18                  = 0.0f, bool filter = true, float gamma = 0.45454545);
19    virtual Colour GetValue(float U, float V, Intersection& HitPoint)
20                      const override;
21
22    Bitmap* BitmapTexture::GetBitmap();
23
24 };
```

Listing 2.91: BitmapTexture source file

```
1 #include "BitmapTexture.h"
2
3 #include <iostream>
4 #include "ToneMapping.h"
5 using namespace std;
6
7 BitmapTexture::BitmapTexture(Bitmap bitmap, float tileU, float tileV,
8     float offsetU, float offsetV, bool filter, float gamma)
9     :
10    tilingU(tileU),
11    tilingV(tileV),
12    OffsetU(offsetU),
13    OffsetV(offsetV),
14    Filter(filter),
15    Gamma(gamma)
16 {
17     BitmapBuffer = *(new Bitmap(bitmap));
18 }
19
20 BitmapTexture::BitmapTexture(std::string filename, float tileU, float
21     tileV, float offsetU, float offsetV, bool filter, float gamma)
22     : BitmapBuffer{ Bitmap::LoadPPM(filename) },
23     tilingU(tileU),
24     tilingV(tileV),
25     OffsetU(offsetU),
26     OffsetV(offsetV),
27     Filter(filter),
28     Gamma(gamma)
29 }
30
31 BitmapTexture::BitmapTexture(std::string filename, string type, float
32     tileU, float tileV, float offsetU, float offsetV, bool filter,
33     float gamma)
34     : BitmapBuffer(Bitmap::LoadPFM(filename)), tilingU(tileU), tilingV(
35         tileV), OffsetU(offsetU), OffsetV(offsetV), Filter(filter),
36         Gamma(gamma)
37
38 Colour BitmapTexture::GetValue(float U, float V, Intersection& HitPoint
39     ) const
40 {
41     float uu = U;
42     float vv = V;
43     U = clamp(fmodf((U + OffsetU) * tilingU, 1.0f), 0.0f, 1.0f);
44     V = clamp(fmodf((V + OffsetV) * tilingV, 1.0f), 0.0f, 1.0f);
```

```
45
46     if (Filter)
47     {
48         U = U * (BitmapBuffer.GetWidth() - 1 - 0.5);
49         V = (1 - V) * (BitmapBuffer.GetHeight() - 1 - 0.5);
50         int u = U;
51         int v = V;
52         Colour uv00(BitmapBuffer.GetPixel(u      , v));
53         Colour uv10(BitmapBuffer.GetPixel(u + 1 , v));
54         Colour uv01(BitmapBuffer.GetPixel(u      , v + 1));
55         Colour uv11(BitmapBuffer.GetPixel(u + 1 , v + 1));
56         return (BilinearInterp<Colour>(U - u, V - v, uv00, uv10, uv01,
57                                         uv11) ^ (1 / Gamma));
58     }
59     return BitmapBuffer.GetPixel(U * (BitmapBuffer.GetWidth() - 1),
60                                  (1 - V) * (BitmapBuffer.GetHeight() - 1)) ^ (1 / Gamma);
60 }
```

2.12.2 CheckerTexture

Listing 2.92: CheckerTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3
4 class CheckerTexture : public Texture
5 {
6 public:
7     CheckerTexture(Texture* c1, Texture* c2)
8         : Colour1(c1), Colour2(c2), tilingU(1), tilingV(1), btile(true)
9     {}
10
11    CheckerTexture(Texture* c1, Texture* c2, float utile, float vtile)
12        : Colour1(c1), Colour2(c2), tilingU(utile), tilingV(vtile),
13          btile(true)
14    {}
15    virtual Colour GetValue(float U, float V, Intersection& HitPoint)
16        const override
17    {
18        if (btile)
19        {
20            int u = U * tilingU * 2;
21            int v = V * tilingV * 2;
22
23            if (((u % 2 == 0) && (v % 2 == 0)) || ((u % 2 == 1) && (v %
24                2 == 1)))
25            {
26                return Colour2->GetValue(U, V, HitPoint);
27            }
28        }
29        else
30        {
31            float u = 0.5 / tilingU;
32            float v = 0.5 / tilingV;
33            if ((U <= 0.5 && U >= 0.5 - u) && (V >= 0.5 && V <= 0.5 +
34                v)) ||
35                ((U >= 0.5 && U <= 0.5 + u) && (V <= 0.5 && V >= 0.5 -
36                v)))
37            return Colour2->GetValue(U, V, HitPoint);
38        }
39        return Colour1->GetValue(U, V, HitPoint);
40    }
41
42 private:
43     Texture* Colour1;
44     Texture* Colour2;
45
46     float tilingU;
47     float tilingV;
48
49     bool btile;
```


2.12.3 DiscTexture

Listing 2.93: DiscTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4 class DiscTexture : public Texture
5 {
6 public:
7     DiscTexture(Texture* map, Texture* base, float radius = 1.0f)
8         :Disc(map), Background(base), Radius(1/radius)
9     {
10         assert(radius <= 1.0f);
11     }
12
13     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
14         const override
15     {
16         Colour c1 = Disc->GetValue(U, V, HitPoint);
17         U = U * Radius * 2;
18         V = V * Radius * 2;
19         U = (Radius - U) * (Radius - U);
20         V = (Radius - V) * (Radius - V);
21
22         if (U + V < 1.0)
23             return c1;
24
25         return Background->GetValue(U, V, HitPoint);
26     }
27 private:
28     Texture* Disc;
29     Texture* Background;
30     float Radius;
31 };
```

2.12.4 GradientTexture

Listing 2.94: GradientTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4
5 // Scaling V > 1.0 will compress
6 // Adding to V will translate down
7 // Subtracting to V will translate up
8 class GradientTexture : public Texture
9 {
10 public:
11     GradientTexture(Texture* colour1, Texture* colour2, Texture*
12                     colour3, float colour2position = 0.25)
13         : Colour1(colour1), Colour2(colour2), Colour3(colour3),
14           Colour2Position(colour2position)
15     {}
16
17     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
18         const override
19     {
20
21         Colour c1 = Colour1->GetValue(U, V, HitPoint);
22         Colour c2 = Colour2->GetValue(U, V, HitPoint);
23         Colour c3 = Colour3->GetValue(U, V, HitPoint);
24
25         if (V >= Colour2Position)
26             return mix(c1, c2, V / (1 - Colour2Position) -
27                         Colour2Position * 1/(1-Colour2Position));
28         return mix(c2, c3, V / (Colour2Position));
29     }
30
31 private:
32     Texture* Colour1;
33     Texture* Colour2;
34     Texture* Colour3;      // The colour prior to this mask texture
35     float Colour2Position;
36 }
```

2.12.5 MaskTexture

Listing 2.95: MaskTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4 class MaskTexture : public Texture
5 {
6 public:
7     MaskTexture(Texture* map, Texture* mask, Texture* base)
8         : Map(map), Mask(mask), Base(base), invertMask(false)
9     {}
10
11    MaskTexture(Texture* map, Texture* mask, Texture* base, bool
12        invertMask)
13        : Map(map), Mask(mask), Base(base), invertMask(invertMask)
14    {}
15
16    virtual Colour GetValue(float U, float V, Intersection& HitPoint)
17        const override
18    {
19        Colour mask = Mask->GetValue(U, V, HitPoint);
20        Colour map = Map->GetValue(U, V, HitPoint);
21        Colour base = Base->GetValue(U, V, HitPoint);
22        float value = (mask.GetR() + mask.GetG() + mask.GetB()) / 3;
23        if (!invertMask)
24            return mix(map, base, value);
25        return mix(map, base, 1 - value);
26    }
27
28 private:
29     Texture* Map;
30     Texture* Mask;
31     Texture* Base;      // The colour prior to this mask texture
32     bool invertMask;
33 };
```

2.12.6 MixTexture

Listing 2.96: MixTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4 #include "SolidColourTexture.h"
5
6 class MixTexture : public Texture
7 {
8 public:
9     MixTexture(Texture* colour1, Texture* colour2, Texture* mix_amount)
10        : Colour1(colour1), Colour2(colour2), MixAmount(mix_amount)
11    {}
12
13     MixTexture(Texture* colour1, Texture* colour2, float mix_percent)
14        : Colour1(colour1), Colour2(colour2)
15    {
16         float percent = (100 - mix_percent) / 100.0;
17         assert(percent >= 0 && percent <= 1.0);
18         MixAmount = new SolidColourTexture(percent, percent, percent);
19    }
20
21
22     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
23         const override
24    {
25         Colour colour1 = Colour1->GetValue(U, V, HitPoint);
26         Colour colour2 = Colour2->GetValue(U, V, HitPoint);
27         Colour mix_amount = MixAmount->GetValue(U, V, HitPoint);
28         float value = (mix_amount.GetR() + mix_amount.GetG() +
29                         mix_amount.GetB()) / 3.0f;
30
31         return mix(colour1, colour2, value);
32    }
33 private:
34     Texture* Map;
35     Texture* Colour1;
36     Texture* Colour2;
37     Texture* MixAmount;
38 };
```

2.12.7 RadialTexture

Listing 2.97: RadialTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4
5 class RadialTexture : public Texture
6 {
7 public:
8     RadialTexture(Texture* c1, Texture* c2, float radius = 1.0f)
9         : Colour1(c1), Colour2(c2), Radius(1 / radius)
10    {
11        assert(radius <= 1.0f);
12    }
13
14     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
15     const override
16    {
17        Colour c1 = Colour1->GetValue(U, V, HitPoint);
18        Colour c2 = Colour2->GetValue(U, V, HitPoint);
19
20        U = U * Radius * 2;
21        V = V * Radius * 2;
22        U = (Radius - U) * (Radius - U);
23        V = (Radius - V) * (Radius - V);
24
25        if (U + V < 1.0)
26            return mix(c1, c2, U + V);
27    }
28 private:
29     Texture* Colour1;
30     Texture* Colour2;
31     float Radius;
32 };
```

2.12.8 SolidColourTexture

Listing 2.98: SolidColourTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3
4 // Solid colour texture
5 class SolidColourTexture : public Texture
6 {
7 public:
8     SolidColourTexture(float r, float g, float b, float gamma =
9                     1.0/2.2)
10    {
11        colour = Colour(r,g,b) ^ (1.0f / gamma);
12    }
13    virtual Colour GetValue(float U, float V, Intersection& HitPoint)
14        const override
15    {
16        return colour;
17    }
18 private:
19     Colour colour;
20
21 };
```

2.12.9 StripeTexture

Listing 2.99: StripeTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../../Core/Utility/Misc.h"
4
5
6 class StripeTexture : public Texture
7 {
8 public:
9     StripeTexture(Texture* colour1, Texture* colour2, float tile = 16,
10                 float rotation = 3.14*2)
11         : Colour1(colour1), Colour2(colour2), Tile(tile), Rotation(
12           rotation)
13     {}
14
15     virtual Colour GetValue(float U, float V, Intersection& HitPoint)
16         const override
17     {
18         float costheta = cosf(Rotation);
19         float sintheta = sinf(Rotation);
20         int u = std::round((U * costheta) - (V * sintheta)) * Tile *
21             2;
22         //int v = ((U * sintheta) + (V * costheta)) * 1.0 * 2;
23         if ((u % 2 == 0))
24         {
25             return Colour2->GetValue(U, V, HitPoint);
26         }
27         return Colour1->GetValue(U, V, HitPoint);
28     }
29
30 private:
31     Texture* Colour1;
32     Texture* Colour2;
33     float Tile;
34     float Rotation;
35 }
```

2.12.10 ToneMapping

Listing 2.100: ToneMapping header file

```
1 #pragma once
2
3 #include "../Colour.h"
4
5 class ToneMapping
6 {
7 protected:
8     virtual Colour Evaluate(const Colour& c) const = 0;
9 };
10
11 class ExposureToneMapper : public ToneMapping
12 {
13 public:
14     ExposureToneMapper(float exposure)
15         : Exposure(exposure)
16     { }
17
18     Colour Evaluate(const Colour& c) const override;
19
20     float Exposure;
21 };
22
23 inline Colour ExposureToneMapper::Evaluate(const Colour& c) const
24 {
25     float gamma = 2.2;
26     Colour mapped(Colour(1) - Colour(exp(-c.GetR() * Exposure), exp(-c.
27         GetG() * Exposure), exp(-c.GetB() * Exposure)));
28     return mapped ^ (1.0f / gamma);
29 }
```

2.12.11 WireframeTexture

Listing 2.101: WireframeTexture header file

```
1 #pragma once
2 #include "../Texture.h"
3 #include "../Bitmap.h"
4 #include <string>
5
6 class WireframeTexture : public Texture
7 {
8 public:
9     WireframeTexture();
10    WireframeTexture(float width, Colour& colour = Colour(1,1,1));
11    virtual Colour GetValue(float U, float V, Intersection& HitPoint)
12        const override;
13    float WireframeTexture::GetWidth();
14 private:
15     float Width;
16     Colour WColour;
17     float Gamma;
18 };
```

Listing 2.102: WireframeTexture source file

```
1 #include "WireframeTexture.h"
2
3 #include <iostream>
4
5 using namespace std;
6
7 WireframeTexture::WireframeTexture()
8 :Width(0.015), WColour(1, 1, 1), Gamma(0.45454545)
9 {}
10
11 WireframeTexture::WireframeTexture(float width, Colour& colour)
12 : Width(width), Gamma(0.45454545)
13 {}
14
15
16 Colour WireframeTexture::GetValue(float U, float V, Intersection&
17 intersection) const
18 {
19     if (intersection.U0 < Width || intersection.V0 < Width ||
20         intersection.U0 + intersection.V0 > 1 - Width)
21     {
22         return Colour(1,1,1);
23     }
24     return Colour(0,0,0);
25 }
```

2.13 Volume

2.13.1 ConstantVolume

Listing 2.103: ConstantVolume header file

```
1 #pragma once
2
3 #include "../Colour.h"
4 #include "Volume.h"
5 #include "../../Core/Math/Transform.h"
6
7 class ConstantVolume : public Volume
8 {
9 public:
10     ConstantVolume(Transform* local_to_world, const Colour& albedo,
11                     const Colour& sigma_a, const Colour& sigma_s, const Colour&
12                     emission, BBox box, PhaseFunction* phase_function = new
13                     Isotropic());
14
15     Colour Absorption(Point v) const override;
16     Colour Emission(Point v) const override;
17     Colour Scattering(Point v) const override;
18     Colour Extinction(Point v) const override;
19     Colour Albedo(Point v) const override;
20     bool IntersectRay(const Ray& ray, VolumeIntersection& intersection)
21         const override;
22     BBox Bounds() const override;
23
24 private:
25     Transform* localToWorld;
26     Transform* worldToLocal;
27     BBox bounds;
28     Colour sigmaA;
29     Colour sigmaS;
30     Colour emission;
31     Colour albedo;
32 };
33
34 inline ConstantVolume::ConstantVolume(Transform* local_to_world, const
35                                         Colour& albedo, const Colour& sigma_a, const Colour& sigma_s, const
36                                         Colour& emission, BBox box, PhaseFunction* phase_function)
37     : Volume(phase_function),
38     localToWorld(local_to_world),
39     worldToLocal(new Transform(local_to_world->Inverse())),
40     sigmaA(sigma_a),
41     sigmaS(sigma_s),
42     emission(emission),
43     bounds(box),
44     albedo(albedo)
45 {
46 }
```

```

41 inline Colour ConstantVolume::Absorption(Point v) const
42 {
43     return sigmaA;
44 }
45
46 inline Colour ConstantVolume::Emission(Point v) const
47 {
48     return emission;
49 }
50
51 inline Colour ConstantVolume::Scattering(Point v) const
52 {
53     return sigmaS;
54 }
55
56 inline Colour ConstantVolume::Extinction(Point v) const
57 {
58     return sigmaA + sigmaS;
59 }
60
61 inline Colour ConstantVolume::Albedo(Point v) const
62 {
63     return albedo;
64 }
65
66 inline bool ConstantVolume::IntersectRay(const Ray& ray,
    VolumeIntersection& intersection) const
67 {
68     Ray localRay = (*worldToLocal) * ray;
69
70     if(bounds.IntersectRay(localRay, intersection.t0, intersection.t1))
71     {
72         intersection.point = ray(intersection.t0);
73         intersection.phaseFunction = phaseFunction;
74         return true;
75     }
76
77     return false;
78 }
79
80 inline BBox ConstantVolume::Bounds() const
81 {
82     return (*localToWorld) * bounds;
83 }

```

2.13.2 PhaseFunction

Listing 2.104: PhaseFunction header file

```
1 #pragma once
2
3 #include "../Intersection.h"
4
5 class PhaseFunction
6 {
7 public:
8     virtual ~PhaseFunction();
9
10    virtual float PDF(const Vector& wo, const Vector& wi) const = 0;
11 };
12
13 class Isotropic : public PhaseFunction
14 {
15 public:
16     virtual float PDF(const Vector& wo, const Vector& wi) const
17         override;
18 };
19
20 class HenyeyGreenstein : public PhaseFunction
21 {
22 public:
23     HenyeyGreenstein(float g);
24     float PDF(const Vector& wo, const Vector& wi) const override;
25 private:
26     float g;
27 };
28
29 class HenyeyGreensteinBlend : public PhaseFunction
30 {
31 public:
32     HenyeyGreensteinBlend(float g1, float g2, float blendAmount);
33     float PDF(const Vector& wo, const Vector& wi) const override;
34 private:
35     float g1;
36     float g2;
37     float blendAmount;
38 };
39 inline PhaseFunction::~PhaseFunction()
40 {
41 }
42
43 inline float Isotropic::PDF(const Vector& wo, const Vector& wi) const
44 {
45     return 1.0f / (4.0f * M_PI);
46 }
47
48
```

```

49 inline HenyeyGreenstein::HenyeyGreenstein(float g)
50     : g(g)
51 {
52
53 }
54
55 inline float HenyeyGreenstein::PDF(const Vector& wo, const Vector& wi)
56     const
57
58     const float cosTheta = Dot(wo, wi);
59     return 1.0f / (4.0f * M_PI) * (1.0f - g * g) / std::powf(1.0f + g *
60         g - 2.0f * g * cosTheta, 1.5f);
61 }
62
63 inline HenyeyGreensteinBlend::HenyeyGreensteinBlend(float g1, float g2,
64     float blendAmount)
65     : g1(g1), g2(g2), blendAmount(blendAmount)
66 {
67
68 inline float HenyeyGreensteinBlend::PDF(const Vector& wo, const Vector&
69     wi) const
70
71     const float cosTheta = Dot(wo, wi);
72     float pdf1 = 1.0f / (4.0f * M_PI) * (1.0f - g1 * g1) / std::powf
73         (1.0f + g1 * g1 - 2.0f * g1 * cosTheta, 1.5f);
74     float pdf2 = 1.0f / (4.0f * M_PI) * (1.0f - g2 * g2) / std::powf
75         (1.0f + g2 * g2 - 2.0f * g2 * cosTheta, 1.5f);
76     return lerp(pdf1, pdf2, blendAmount);
77 }
```

2.13.3 RayMarcher

Listing 2.105: RayMarcher header file

```
1 #pragma once
2
3 #include "Volume.h"
4 #include "VolumeIntegrator.h"
5 #include "../Colour.h"
6 #include "../Scene.h"
7 #include "../Sampler.h"
8
9 class RayMarcher : public VolumeIntegrator
10 {
11
12 public:
13     explicit RayMarcher(float step_size)
14         : stepSize(step_size)
15     {
16     }
17     VolumeIntegrationResult Li(const Scene* scene, const Ray& ray,
18         const Sampler* sampler) const override;
18     Colour Transmittance(const Scene* scene, const Ray& ray, const
19         Sampler* sampler) const override;
20
21     static bool IsInShadow(const Scene* scene, const Ray& ray, Light*
22         light, Point& lightPos);
23
24 protected:
25     float stepSize;
26
27 };
28
29 inline VolumeIntegrationResult RayMarcher::Li(const Scene* scene, const
30     Ray& ray, const Sampler* sampler) const
31 {
32     VolumeIntersection vi;
33     VolumeIntegrationResult result;
34     if (scene->SVolumes == nullptr || !scene->SVolumes->IntersectRay(
35         ray, vi))
36     {
37         result.Transmittance = Colour(1.0f);
38         result.Luminance = Colour(0.0f);
39         return result;
40     }
41     // Check for any geometry blocking the volume
42     Intersection ii;
43     if (scene->Intersect(ray, ii) && (ii.GetPoint() - ray.GetOrigin()).LengthSquared() < (vi.point - ray.GetOrigin()).LengthSquared())
44     {
45         result.Transmittance = Colour(1.0f);
46         result.Luminance = Colour(0.0f);
47         return result;
48     }
49 }
```

```

44
45
46     float t0 = vi.t0;
47     float t1 = vi.t1;
48     Volume* volume = scene->SVolumes;
49     // The number of integration steps
50     int numberOfSteps = int(ceil((vi.t1 - vi.t0) / stepSize));
51
52     // Calculate the step size dt
53     float dt = (vi.t1 - vi.t0) / numberOfSteps;
54     Vector stepDirection = ray.GetDirection();
55     Point position = ray(vi.t0);
56
57     Colour transmission(1.0f);
58     Colour luminance(0.0f);
59     t0 += dt;
60
61     while (t0 < t1)
62     {
63
64         position = position + stepDirection * dt;
65         Colour Le = volume->Emission(position);
66         Colour sigma_a = volume->Absorption(position);
67
68         Colour dTr = Colour::Exp(-sigma_a * stepSize);
69         transmission *= dTr;
70         luminance += transmission * Le;
71
72         // if transmittance becomes negligible we terminate early
73         if (transmission.Average() < 1e-6f)
74         {
75             const float probabilityOfStopping = sampler->Sample1D();
76             const float shouldContinue = probabilityOfStopping > 0.5f;
77             if (shouldContinue)
78             {
79                 break;
80             }
81             transmission /= (1 - probabilityOfStopping);
82         }
83
84         // calculate contribution from scattering
85         Colour sigma_s = volume->Scattering(position);
86         Colour dTs = Colour::Exp(-sigma_s * stepSize);
87         transmission *= dTs;
88         for (auto light : scene->GetLights())
89         {
90             Vector wi;
91             Point pointOnLight;
92             Intersection intersection;
93             intersection.IPoint = position;
94             Colour L = light->SampleLight(sampler->Sample2D(),
pointOnLight, wi, intersection);

```

```

95     Ray shadowRay(position, wi, 0, t1);
96     if (L != Colour(0.0f) && !IsInShadow(scene, shadowRay,
97         light, pointOnLight))
98     {
99         Colour Ld = L * Transmittance(scene, shadowRay, sampler
100            );
101        luminance += volume->Albedo(position) * transmission *
102            Ld * sigma_s * volume->GetPhaseFunction()->PDF(
103                stepDirection, wi);
104    }
105
106    result.Transmittance = transmission;
107    result.Luminance = luminance;
108    return result;
109 }
110
111 inline Colour RayMarcher::Transmittance(const Scene* scene, const Ray&
112     ray, const Sampler* sampler) const
113 {
114     if (scene->GetVolume() == nullptr)
115     {
116         return Colour(1.0f);
117     }
118     float step, offset;
119     step = 4.0f * stepSize;
120     offset = sampler->Sample1D();
121     float length = ray.GetDirection().Length();
122     Ray rn = Ray(ray.GetOrigin(), ray.GetDirection() / length, ray.
123         MinT * length, ray.MaxT * length);
124     auto volume = scene->GetVolume();
125     VolumeIntersection vi;
126     if (!volume->IntersectRay(rn, vi))
127     {
128         return Colour(1.0f);
129     }
130     // Calculate optical thickness
131     Colour tau(0.0f);
132     vi.t0 += offset * step;
133     while (vi.t0 < vi.t1)
134     {
135         tau += volume->Extinction(rn(vi.t0));
136         vi.t0 += step;
137     }
138     // calculate transmittance
139     return Colour::Exp(-tau * step);

```

```
140 inline bool RayMarcher::IsInShadow(const Scene* scene, const Ray& ray,
141     Light* light, Point& lightPos)
142 {
143     Intersection is;
144     float tCurr = INFINITY;
145     if (scene->GetAccelerator()->Intersect(ray, tCurr, is, true))
146     {
147         float distanceToLight = (lightPos - ray.GetOrigin()).Length();
148         float distance = (is.GetPoint() - ray.GetOrigin()).Length();
149         return (distance) < (distanceToLight);
150     }
151     return false;
152 }
```

2.13.4 Volume

Listing 2.106: Volume header file

```
1 #pragma once
2 #include "../BBox.h"
3 #include "PhaseFunction.h"
4
5 class Colour;
6 class Ray;
7 struct Intersection;
8
9 struct VolumeIntersection
10 {
11     float t0, t1;
12     float stepLength;
13     PhaseFunction* phaseFunction;
14     Point point;
15 };
16
17 class Volume
18 {
19
20 public:
21
22     explicit Volume(PhaseFunction* phase_function);
23     virtual ~Volume();
24     // Returns the absorption coefficient for a given point
25     virtual Colour Absorption(Point p) const = 0;
26     // Returns the emission coefficient for a given point
27     virtual Colour Emission(Point p) const = 0;
28     // Returns the scattering coefficient for a given point
29     virtual Colour Scattering(Point p) const = 0;
30     // Returns the extinction coefficient for a given point
31     virtual Colour Extinction(Point p) const = 0;
32     // Returns the albedo of the volume for a given point
33     virtual Colour Albedo(Point v) const = 0;
34     virtual bool IntersectRay(const Ray& ray, VolumeIntersection&
35         intersection) const = 0;
36     virtual BBox Bounds() const = 0;
37     PhaseFunction* GetPhaseFunction() const;
38
39 protected:
40     PhaseFunction* phaseFunction;
41 };
42
43 inline Volume::Volume(PhaseFunction* phase_function)
44     : phaseFunction(phase_function)
45 {
46
47 }
```

```
49 inline Volume::~Volume()
50 {
51
52 }
53
54 inline PhaseFunction* Volume::GetPhaseFunction() const
55 {
56     return phaseFunction;
57 }
```

2.13.5 VolumeData

Listing 2.107: VolumeData header file

```
1 #pragma once
2
3 #include "../BBox.h"
4 #include <vector>
5 #include <fstream>
6
7 struct VolumeData
8 {
9     int nx;
10    int ny;
11    int nz;
12    BBox bounds;
13    std::vector<float> density;
14    std::vector<float> temperature;
15
16    friend std::istream& operator>>(std::istream& is, VolumeData &data)
17    {
18        is >> data.nx >> data.ny >> data.nz;
19        is >> data.bounds;
20        int numberDensities;
21        is >> numberDensities;
22        int numberTemperatures;
23        is >> numberTemperatures;
24        float temp;
25        for (int i = 0; i < numberDensities; ++i)
26        {
27            is >> temp;
28            data.density.push_back(temp);
29        }
30
31        for (int i = 0; i < numberTemperatures; ++i)
32        {
33            is >> temp;
34            data.temperature.push_back(temp);
35        }
36
37        return is;
38    }
39 };
40
41 inline void LoadVolumeData(VolumeData& data, char* filename)
42 {
43     std::ifstream file(filename);
44     if (!file)
45     {
46         std::cerr << "Cannot load volume data: " << filename << std::endl;
47     }
48 }
```

```
49     file >> data;  
50     file.close();  
51 }
```

2.13.6 Volumelntegrator

Listing 2.108: VolumeIntegrator header file

```
1 #pragma once
2
3 class Sampler;
4 class Ray;
5 class Scene;
6 class Colour;
7
8 struct VolumeIntegrationResult
9 {
10     VolumeIntegrationResult();
11     VolumeIntegrationResult(Colour L, Colour T);
12     Colour Luminance;
13     Colour Transmittance;
14 };
15
16 inline VolumeIntegrationResult::VolumeIntegrationResult()
17     : Transmittance(1.0f),
18     Luminance(0.0f)
19 {
20
21 }
22
23 inline VolumeIntegrationResult::VolumeIntegrationResult(Colour L,
24     Colour T)
25     : Luminance(L),
26     Transmittance(T)
27 {
28
29 class VolumeIntegrator
30 {
31 public:
32     virtual ~VolumeIntegrator();
33
34     virtual VolumeIntegrationResult Li(const Scene* scene, const Ray&
35         ray, const Sampler* sampler) const = 0;
36     virtual Colour Transmittance(const Scene* scene, const Ray& ray,
37         const Sampler* sampler) const = 0;
38 };
39
40 inline VolumeIntegrator::~VolumeIntegrator()
41 {
```

2.13.7 VoxelBuffer

Listing 2.109: VoxelBuffer header file

```
1 #pragma once
2 #include <vector>
3 #include "../Core/Math/Transform.h"
4
5 class VoxelBuffer
6 {
7     VoxelBuffer(int size_x, int size_y, int size_z, Transform*
8                 local_to_world)
9         : sizeX(size_x),
10           sizeY(size_y),
11           sizeZ(size_z),
12           localToWorld(local_to_world),
13           data(sizeX * sizeY * sizeZ)
14     {}
15
16 public:
17     float operator()(int x, int y, int z) const;
18     void operator()(int x, int y, int z, float value);
19
20 protected:
21     int sizeX, sizeY, sizeZ;
22     Transform* localToWorld;
23     std::vector<float> data;
24
25 private:
26     int index(int x, int y, int z) const;
27 };
28
29 inline float VoxelBuffer::operator()(int x, int y, int z) const
30 {
31     return data[index(x, y, z)];
32 }
33
34 inline void VoxelBuffer::operator()(int x, int y, int z, float value)
35 {
36     data[index(x, y, z)] = value;
37 }
38
39 inline int VoxelBuffer::index(int x, int y, int z) const
40 {
41     return (x + y * sizeX + z * sizeX * sizeY);
42 }
```

2.13.8 VoxelGrid

Listing 2.110: VoxelGrid header file

```
1 #pragma once
2 #include "Volume.h"
3 #include <vector>
4 #include "../Colour.h"
5
6 #include "VolumeData.h"
7 #include "../../Core/Math/Transform.h"
8
9 class VolumeGrid : public Volume
10 {
11 public:
12     VolumeGrid(Transform* local_to_world, const Colour& albedo, const
13         Colour& sigma_a, const Colour& sigma_s, const Colour& emission,
14         const Colour& temperature, VolumeData& data, PhaseFunction*
15         phase_function = new Isotropic(), float scale = 1.0f);
16     Colour Absorption(Point p) const override;
17     Colour Emission(Point p) const override;
18     Colour Scattering(Point p) const override;
19     Colour Extinction(Point p) const override;
20     Colour Albedo(Point v) const override;
21     float Temperature(Point p) const;
22     float Density(Point p) const;
23     bool IntersectRay(const Ray& ray, VolumeIntersection& intersection)
24         const override;
25     BBox Bounds() const override;
26
27     int Index(int x, int y, int z) const;
28 private:
29     BBox bounds;
30     Colour albedo;
31     Colour sigma_a;
32     Colour sigma_s;
33     Colour emission;
34     Colour temperature;
35     int sizeX, sizeY, sizeZ;
36     float densityScale;
37     Transform* localToWorld;
38     Transform* worldToLocal;
39     std::vector<float> densities;
40     std::vector<float> temperatures;
41 };
42
43 inline VolumeGrid::VolumeGrid(Transform* local_to_world, const Colour&
44     albedo, const Colour& sigma_a, const Colour& sigma_s, const Colour&
45     emission, const Colour& temperature, VolumeData& data,
46     PhaseFunction* phase_function, float scale)
47     : Volume(phase_function),
48     albedo(albedo),
```

```

43     sigma_a(sigma_a),
44     sigma_s(sigma_s),
45     emission(emission),
46     temperature(temperature),
47     densities(data.density),
48     temperatures(data.temperature),
49     localToWorld(local_to_world),
50     worldToLocal(new Transform(localToWorld->Inverse())),
51     sizeX(data.nx),
52     sizeY(data.ny),
53     sizeZ(data.nz),
54     bounds(data.bounds),
55     densityScale(scale)
56 {
57
58 }
59
60 inline Colour VolumeGrid::Absorption(Point p) const
61 {
62     return Density(*worldToLocal * p) * sigma_a;
63 }
64
65 inline Colour VolumeGrid::Emission(Point p) const
66 {
67     p = *worldToLocal * p;
68     return Temperature(p) * temperature + Density(p) * emission;
69 }
70
71 inline Colour VolumeGrid::Scattering(Point p) const
72 {
73     return Density(*worldToLocal * p) * sigma_s;
74 }
75
76 inline Colour VolumeGrid::Extinction(Point p) const
77 {
78     return Density(*worldToLocal * p) * (sigma_a + sigma_s);
79 }
80
81 inline Colour VolumeGrid::Albedo(Point v) const
82 {
83     return albedo;
84 }
85
86 inline float VolumeGrid::Temperature(Point p) const
87 {
88     if (!bounds.IsInside(p))
89     {
90         return 0.0f;
91     }
92
93     // convert from world space to voxel space [0,0,0] - [1,1,1]
94     Vector voxel = bounds.RelativePosition(p);

```

```

95 // convert to lower index of voxel grid
96 voxel[0] = voxel[0] * sizeX - 0.5;
97 voxel[1] = voxel[1] * sizeY - 0.5;
98 voxel[2] = voxel[2] * sizeZ - 0.5;
99 int vx = int(floor(voxel[0]));
100 int vy = int(floor(voxel[1]));
101 int vz = int(floor(voxel[2]));
102
103 float dx = voxel[0] - vx;
104 float dy = voxel[1] - vy;
105 float dz = voxel[2] - vz;
106 // interpolate across x-axis
107 float c00 = lerp(temperatures[Index(vx, vy, vz)],
108                   temperatures[Index(vx + 1, vy, vz)], dx);
109 float c10 = lerp(temperatures[Index(vx, vy + 1, vz)],
110                   temperatures[Index(vx + 1, vy + 1, vz)], dx);
111 float c01 = lerp(temperatures[Index(vx, vy, vz + 1)],
112                   temperatures[Index(vx + 1, vy, vz + 1)], dx);
113 float c11 = lerp(temperatures[Index(vx, vy + 1, vz + 1)],
114                   temperatures[Index(vx + 1, vy + 1, vz + 1)], dx);
115 // interpolate across y-axis
116 float c0 = lerp(c00, c10, dy);
117 float c1 = lerp(c01, c11, dy);
118 // interpolate across z-axis
119 return lerp(c0, c1, dz);
120 }
121
122 inline float VolumeGrid::Density(Point p) const
123 {
124     if (!bounds.IsInside(p))
125     {
126         return 0.0f;
127     }
128
129     // convert from world space to voxel space [0,0,0] - [1,1,1]
130     Vector voxel = bounds.RelativePosition(p);
131     // convert to lower index of voxel grid
132     voxel[0] = voxel[0] * sizeX - 0.5;
133     voxel[1] = voxel[1] * sizeY - 0.5;
134     voxel[2] = voxel[2] * sizeZ - 0.5;
135     int vx = int(floor(voxel[0]));
136     int vy = int(floor(voxel[1]));
137     int vz = int(floor(voxel[2]));
138
139     // Trilinear interpolation
140     //   c001 .----. c11----. c111
141     //   / |     /       / |
142     //   / |     . c1     / |
143     // c001 .--|----|----. / |
144     //   |     | c01 |     | |
145     //   |     |---|_c10_|---. c110

```

```

143     //      | /      | /      | /
144     //      | /      /c0      | /
145     //      |/_----/_----_|/
146     //      c000      c00      c100
147
148     float dx = voxel[0] - vx;
149     float dy = voxel[1] - vy;
150     float dz = voxel[2] - vz;
151     // interpolate across x-axis
152     float c00 = lerp(densities[Index(vx, vy, vz)], densities[
153         Index(vx + 1, vy, vz)], dx);
154     float c10 = lerp(densities[Index(vx, vy + 1, vz)], densities[
155         Index(vx + 1, vy + 1, vz)], dx);
156     float c01 = lerp(densities[Index(vx, vy, vz + 1)], densities[
157         Index(vx + 1, vy, vz + 1)], dx);
158     float c11 = lerp(densities[Index(vx, vy + 1, vz + 1)], densities[
159         Index(vx + 1, vy + 1, vz + 1)], dx);
160     // interpolate across y-axis
161     float c0 = lerp(c00, c10, dy);
162     float c1 = lerp(c01, c11, dy);
163     // interpolate across z-axis
164     return densityScale * lerp(c0, c1, dz);
165 }
166
167 inline bool VolumeGrid::IntersectRay(const Ray& ray, VolumeIntersection
168     & intersection) const
169 {
170     Ray localRay = (*worldToLocal) * ray;
171     localRay.MinT = ray.MinT;
172     localRay.MaxT = ray.MaxT;
173     if (bounds.IntersectRay(localRay, intersection.t0, intersection.t1))
174     {
175         intersection.point = ray(intersection.t0);
176         intersection.phaseFunction = phaseFunction;
177         return true;
178     }
179     return false;
180 }
181
182 inline BBox VolumeGrid::Bounds() const
183 {
184     return *localToWorld * bounds;
185 }
186
187 inline int VolumeGrid::Index(int x, int y, int z) const
188 {
189     x = clamp(x, 0, sizeX - 1);
190     y = clamp(y, 0, sizeY - 1);

```

```
189     z = clamp(z, 0, sizeZ - 1);  
190     return (x + y * sizeX + z * sizeX * sizeY);  
191 }
```

3 Miscellaneous

3.1 Globals

Listing 3.1: Globals header file

```
1 #define DEBUG 1
2 #define DEBUG_CAMERA_RAY 0
3 #define FINFINITY std::numeric_limits<float>::infinity()
```

3.2 Main

Listing 3.2: Main source file

```
1 #include "Core/Math/Statistics.h"
2 #include "Core/Utility/RNG.h"
3
4 int main(int* argc, char** argv)
5 {
6     //std::vector<float> f(6, 1 / 6.0);
7     //f[0] = 1;
8     //f[1] = 1;
9     //f[2] = 1;
10    //f[3] = 1;
11    //f[4] = 1;
12    //f[5] = 1;
13    //Distribution1D dist(f);
14    //float x;
15    //float pdf;
16    //for (int i = 0; i < 7; ++i)
17    //{
18    //
19    //    float index = i / 6.0f;
20    //    dist.Sample1D(index, x, pdf);
21    //    std::cout << "sample(" << index << ") ==" << x << std::endl;
22    //    std::cout << pdf << std::endl;
23    //}
24    Bitmap bitmap(Bitmap::LoadPPM("C:\\\\Users\\\\k1332225\\\\Pictures\\\\face.
25     ppm"));
26    //Bitmap bitmap(Bitmap::LoadPFM("I:\\\\temp\\\\hdr\\\\provwash-envmap.pfm
27     "));
28    //std::vector<std::vector<float>> f;
29    //for (int i = 0; i < bitmap.GetHeight(); ++i)
30    //{
31    //    std::vector<float> g;
32    //    for (int j = 0; j < bitmap.GetWidth(); ++j)
33    //    {
34    //        Colour c(bitmap.GetPixel(j, i));
35    //        //std::cout << (c.GetR() + c.GetG() + c.GetB()) << std::
36    //        endl;
37    //        g.push_back(c.GetG() + c.GetB() + c.GetR());
38    //    }
39    //    f.push_back(g);
40    //}
41    //Distribution2D dist2D(f);
42
43    Distribution2D dist2D(&bitmap);
44
45    Bitmap b(bitmap.GetWidth(), bitmap.GetHeight());
46
47    //for (int i = 0; i < bitmap.GetHeight(); ++i)
48    //{
49    //    for (int j = 0; j < bitmap.GetWidth(); ++j)
```

```

47     //  {
48     //      double c = dist2D.rowDistributions[i].PDF[j];
49     //      //std::cout << c << std::endl;
50     //      //c *= 100;
51     //      b.SetPixel(j, i, Colour(c, c, c));
52     //      //std::cout << b.GetPixel(j, i);
53     //  }
54 //}
55
56
57
58 Random::RandomDouble rng(0, 1, 4245123);
59 int samples = 1000000;
60 float dx = 0.0001;
61 for (int i = 0; i < samples; ++i)
62 {
63
64     Point2D uv;
65     float pdf;
66     dist2D.Sample2D(Point2D(rng(), rng()), uv, pdf);
67     pdf = pdf * (bitmap.GetWidth() * bitmap.GetHeight()) / (2 *
68                 M_PI * M_PI * sinf(uv[1]));
69     //std::cout << pdf << std::endl;
70     int u = uv[0] * bitmap.GetWidth();
71     int v = uv[1] * bitmap.GetHeight();
72     Colour c(b.GetPixel(u, v) + Colour(dx, dx, dx));
73     b.SetPixel(u, v, c);
74 }
75 b.SavePPM("C:\\\\Users\\\\k1332225\\\\Pictures\\\\dist.ppm");
76 return 0;
77 }
```

4 Viewer

4.0.1 EmptyRenderer

Listing 4.1: MeshLoader header file

```
1 #include "../PhotonMapping/Engine/Integrators/DirectLightingIntegrator.h"
2 #include "../PhotonMapping/Engine/Integrators/PathTracingIntegrator.h"
3 #include "../PhotonMapping/Engine/Integrators/PhotonMappingIntegrator.h"
4
5 #include "../PhotonMapping/Engine/Light/DirectedAreaLight.h"
6 #include "../PhotonMapping/Engine/Volume/RayMarcher.h"
7 #include "../PhotonMapping/Engine/Volume/ConstantVolume.h"
8 #include "../PhotonMapping/Engine/Volume/VoxelGrid.h"
9 #include "../PhotonMapping/Engine/Renderer.h"
10 #include "../PhotonMapping/Engine/Camera/PinholeCamera.h"
11 #include "../PhotonMapping/Engine/Renderers/TileRenderer.h"
12 #include "../PhotonMapping/Engine/Integrators/WhittedIntegrator.h"
13
14
15 inline Renderer* CreateRenderer()
16 {
17
18     Sampler* sampler = new RandomSampler(1);
19     sampler->GenerateSamples();
20     float Width = 1024;
21     float Height = 768;
22     Bitmap* bitmap = new Bitmap(Width, Height);
23     float AspectRatio = Width / Height;
24
25     Matrix m = XYZTransformMaya(Vector(0.931722568679, 0.766908196234,
26                                         20.5181242177),
27                               Vector(-2.1383527296, 2.6, 6.218422294e-18));
28
29     Camera* camera = new PinholeCamera(m, 39.3077, AspectRatio, 0.1f,
30                                       1000.0f);
31
32     std::vector<Light*> lights;
33     std::vector<Primitive*> geom;
34     Integrator* integrator = new WhittedIntegrator(1);
35     Scene* scene = new Scene(geom, lights, integrator, new BVH(geom),
36                             camera);
37     return new TileRenderer(scene, camera, integrator, bitmap, sampler,
38                            32);
```


4.0.2 Global

Listing 4.2: Global header file

```
1 #pragma once
2 #include "../PhotonMapping/Globals.h"
3 #include "../PhotonMapping/Core/Math/Vector.h"
4 #include "../PhotonMapping/Core/Math/Point.h"
5 #include "../PhotonMapping/Core/Utility/Misc.h"
6 #include "../PhotonMapping/Core/Utility/MeshLoader.h"
7
8 #include "../PhotonMapping/Engine/Ray.h"
9 #include "../PhotonMapping/Engine/Colour.h"
10 #include "../PhotonMapping/Engine/Bitmap.h"
11
12 #include "../PhotonMapping/Engine/Geometry/Sphere.h"
13 #include "../PhotonMapping/Engine/Geometry/Triangle.h"
14 #include "../PhotonMapping/Engine/Geometry/Plane.h"
15 #include "../PhotonMapping/Engine/Geometry/Box.h"
16 #include "../PhotonMapping/Engine/Geometry/Disc.h"
17 #include "../PhotonMapping/Engine/Geometry/Quadrilateral.h"
18
19 #include "../PhotonMapping/Engine/Camera/PinholeCamera.h"
20 #include "../PhotonMapping/Engine/Camera/OrthographicCamera.h"
21 #include "../PhotonMapping/Engine/Camera/ThinLensCamera.h"
22
23 #include "../PhotonMapping/Engine/Light/PointLight.h"
24 #include "../PhotonMapping/Engine/Light/SpotLight.h"
25 #include "../PhotonMapping/Engine/Light/Dome.h"
26 #include "../PhotonMapping/Engine/Light/DistantDiscLight.h"
27
28 #include "../PhotonMapping/Engine/Textures/BitmapTexture.h"
29 #include "../PhotonMapping/Engine/Textures/WireframeTexture.h"
30 #include "../PhotonMapping/Engine/Textures/CheckerTexture.h"
31 #include "../PhotonMapping/Engine/Textures/SolidColourTexture.h"
32 #include "../PhotonMapping/Engine/Textures/MaskTexture.h"
33 #include "../PhotonMapping/Engine/Textures/GradientTexture.h"
34 #include "../PhotonMapping/Engine/Textures/StripeTexture.h"
35 #include "../PhotonMapping/Engine/Textures/DiscTexture.h"
36 #include "../PhotonMapping/Engine/Textures/MixTexture.h"
37 #include "../PhotonMapping/Engine/Textures/RadialTexture.h"
38
39 #include "../PhotonMapping/Engine/Sampler/RandomSampler.h"
40 #include "../PhotonMapping/Engine/Sampler/StratifiedSampler.h"
41
42 #include "../PhotonMapping/Engine/Scene.h"
43
44 #include "../PhotonMapping/Core/Math/Transform.h"
45
46 #include <iostream>
47 #include <chrono>
48 #include <future>
49 #include <omp.h>
```

```
50
51 #include "../PhotonMapping/Core/Utility/RNG.h"
52 #include "../PhotonMapping/Core/Utility/Timer.h"
53 #include "../PhotonMapping/Engine/Accelerators/BVH.h"
54 #include "../PhotonMapping/Engine/Renderers/SimpleRenderer.h"
55 #include "../PhotonMapping/Engine/Integrators/WhittedIntegrator.h"
56 #include "../PhotonMapping/Engine/Integrators/AVIntegrator.h"
57 #include "../PhotonMapping/Engine/Integrators/NormalIntegrator.h"
58 #include "../PhotonMapping/Engine/Integrators/FunctionIntegrator.h"
59
60 #include "../PhotonMapping/Engine/Material.h"
61 #include "../PhotonMapping/Engine/Materials/PhongMaterial.h"
62 #include "../PhotonMapping/Engine/Materials/DiffuseMaterial.h"
63 #include "../PhotonMapping/Engine/Materials/SpecularMaterial.h"
64 #include "../PhotonMapping/Engine/Materials/Plastic.h"
65 #include "../PhotonMapping/Engine/Materials/DielectricMaterial.h"
66
67
68 #include "../PhotonMapping/Engine/Renderers/FunctionRenderer.h"
69 #include "../PhotonMapping/Engine/Renderers/TileRenderer.h"
70
71 #include "../PhotonMapping/Engine/Primitives/GeometricPrimitive.h"
72 #include "../PhotonMapping/Engine/Primitives/InstancePrimitive.h"
```

4.0.3 Main

Listing 4.3: main source file

```
1 #include "viewer.h"
2 #include <QtWidgets/QApplication>
3 #include <QtCore>
4 #include "Global.h"
5 #include "EmptyRenderer.h"
6
7 extern Renderer* CreateRenderer();
8
9 void hello(QString name)
10 {
11     qDebug() << "Hello" << name << "from" << QThread::currentThread();
12 }
13
14
15 int main(int argc, char *argv[])
16 {
17     Renderer* renderer = CreateRenderer();
18 #if 1
19     QApplication a(argc, argv);
20     Viewer w(renderer);
21     auto handler = &RenderThread::UpdateScreen;
22     // x, y, w, h, bitmap
23     auto func = std::bind(handler, w.thread, std::placeholders::_1, std
24                           ::placeholders::_2, std::placeholders::_3, std::placeholders::
25                           _4, std::placeholders::_5);
26     auto start_handler = &RenderThread::StartScreen;
27     auto start_func = std::bind(start_handler, w.thread, std::
28                               placeholders::_1, std::placeholders::_2, std::placeholders::_3,
29                               std::placeholders::_4);
30     renderer->SetOnUpdateCallback(func);
31     renderer->SetOnStartCallback(start_func);
32     qRegisterMetaType<QRgb>("QRgb");
33     w.show();
34     return a.exec();
35 #else
36     timer.Start();
37     renderer->Render();
38     qDebug() << "\n";
39     timer.Stop();
40     qDebug() << "Rendering: " << timer.GetElapsedSeconds() << " seconds
41             ";
42     bitmap->SavePPM("I:\\render.ppm");
43     //bitmap->SavePBM("I:\\render.pbm");
44     return 0;
45 #endif
46
47     //void (RenderThread::*pmfnP)(int, int) = &RenderThread::
48     //    UpdateScreen;
49 }
```

```
44 //w.image = QImage(Width, Height, QImage::Format_RGB888);  
45 //QFuture<void> f1 = run(hello, QString("Alice"));  
46 //f1.waitForFinished();  
47  
48  
49 }
```

4.0.4 RenderThread

Listing 4.4: RenderThread header file

```
1 #ifndef RENDERTHREAD_H
2 #define RENDERTHREAD_H
3
4 #include <QtCore>
5 #include <string>
6 #include <QImage>
7 #include <QRgb>
8 #include "../PhotonMapping/Engine/Bitmap.h"
9
10 #include "../PhotonMapping/Globals.h"
11 #include "../PhotonMapping/Core/Math\Vector.h"
12 #include "../PhotonMapping/Core/Math\Point.h"
13 #include "../PhotonMapping/Core\Utility\Misc.h"
14 #include "../PhotonMapping/Core\Utility\MeshLoader.h"
15
16 #include "../PhotonMapping/Engine\Ray.h"
17 #include "../PhotonMapping/Engine\Colour.h"
18 #include "../PhotonMapping/Engine\Bitmap.h"
19
20 #include "../PhotonMapping/Engine\Geometry\Sphere.h"
21 #include "../PhotonMapping/Engine\Geometry\Triangle.h"
22 #include "../PhotonMapping/Engine\Geometry\Plane.h"
23 #include "../PhotonMapping/Engine\Geometry\Box.h"
24
25 #include "../PhotonMapping/Engine\Camera\PinholeCamera.h"
26 #include "../PhotonMapping/Engine\Camera\OrthographicCamera.h"
27
28 #include "../PhotonMapping/Engine\Light\PointLight.h"
29 #include "../PhotonMapping/Engine\Light\Dome.h"
30
31 #include "../PhotonMapping/Engine/Textures/BitmapTexture.h"
32 #include "../PhotonMapping/Engine/Textures/WireframeTexture.h"
33 #include "../PhotonMapping/Engine/Textures/CheckerTexture.h"
34 #include "../PhotonMapping/Engine\Textures\SolidColourTexture.h"
35 #include "../PhotonMapping/Engine\Textures\MaskTexture.h"
36 #include "../PhotonMapping/Engine\Textures\GradientTexture.h"
37 #include "../PhotonMapping/Engine\Textures\StripeTexture.h"
38 #include "../PhotonMapping/Engine\Textures\DiscTexture.h"
39 #include "../PhotonMapping/Engine\Textures\MixTexture.h"
40 #include "../PhotonMapping/Engine\Textures\RadialTexture.h"
41
42 #include "../PhotonMapping/Engine\Sampler\RandomSampler.h"
43 #include "../PhotonMapping/Engine\Sampler\StratifiedSampler.h"
44
45 #include "../PhotonMapping/Engine\Scene.h"
46
47 #include <iostream>
48 #include <chrono>
```

```

50 #include "../PhotonMapping/Core\Utility\RNG.h"
51 #include "../PhotonMapping/Core\Utility\Timer.h"
52 #include <future>
53 #include <omp.h>
54 #include "../PhotonMapping/Engine\Accelerators\BVH.h"
55 #include "../PhotonMapping/Engine\Renderers\SimpleRenderer.h"
56 #include "../PhotonMapping/Engine\Integrators\WhittedIntegrator.h"
57
58 #include "../PhotonMapping/Engine\Material.h"
59 #include "../PhotonMapping/Engine\Materials\PhongMaterial.h"
60 #include <QWidget>
61 #include <QPainter>
62 #include <QMutex>
63 #include <QRgb>
64 using namespace std;
65
66
67
68 class RenderThread : public QThread
69 {
70     Q_OBJECT
71 public:
72     RenderThread(QObject* parent, Renderer* b, QImage* im);
73     void UpdateScreen(int x, int y, int w, int h, Bitmap* bitmap);
74     void StartScreen(int x, int y, int w, int h);
75     void run() override;
76     Renderer* renderer;
77     QImage* image;
78     QObject* parent;
79     QPainter* qPainter;
80     QMutex mutex;
81
82 signals:
83     void PixelAdded(int x, int y, int w, int h);
84 public slots:
85 };
86
87 inline void RenderThread::StartScreen(int x, int y, int w, int h)
88 {
89     QWidget* a = (QWidget*)parent;
90     qPainter->setPen(QColor(255,0,0));
91     float margin = 1;
92     float bottom = y + h - margin;
93     float right = x + w - margin;
94     float left = x + margin;
95     float top = y + margin;
96     float length = h / 8;
97     float minimumHeight = 15;
98     float minimumWidth = 15;
99     if (h > minimumHeight && w > minimumWidth)
100    {
101        qPainter->drawLine(QPointF(left, top), QPointF(right,top));

```

```

102     qPainter->drawLine(QPointF(left, top), QPointF(left, top +
103                             length));
104     qPainter->drawLine(QPointF(right,top), QPointF(right,top +
105                             length));
106
107     qPainter->drawLine(QPointF(left , bottom), QPointF(right ,
108                             bottom));
109     qPainter->drawLine(QPointF(left , bottom), QPointF(left ,
110                             bottom - length));
111     qPainter->drawLine(QPointF(right , bottom), QPointF(right ,
112                             bottom - length));
113 }
114
115 inline void RenderThread::UpdateScreen(int x, int y, int w, int h,
116                                         Bitmap* bitmap)
117 {
118     QWidget* a = (QWidget*)parent;
119     for(int i = x; i < x + w; ++i)
120     {
121         for(int j = y; j < y + h; ++j)
122         {
123
124             Colour c =bitmap->GetPixel(i,j);
125             image->setPixel(i, j, qRgb(c[0] * 255, c[1] * 255, c
126                                         [2]*255) );
127         }
128     }
129     a->update();
130 }
131
132 #endif // RENDERTHREAD_H

```

Listing 4.5: RenderThread source file

```
1 #include <QtCore>
2 #include "RenderThread.h"
3 #include "Global.h"
4
5 RenderThread::RenderThread(QObject* parent, Renderer* b, QImage* im)
6     :renderer(b), image(im), qPainter(new QPainter(image))
7 {
8     this->parent = parent;
9 }
10
11
12 void RenderThread::run()
13 {
14     QTime t;
15     t.start();
16     qDebug() << "Starting render";
17     renderer->Render();
18     qDebug() << "time taken to render:" << t.elapsed() / 1000.0 << "
19     seconds";
}
```

4.0.5 Viewer

Listing 4.6: viewer header file

```
1 #ifndef VIEWER_H
2 #define VIEWER_H
3
4 #include <QtWidgets/QMainWindow>
5 #include <QtGui>
6 #include <QtCore>
7 #include <QImage>
8 #include <QFileDialog>
9 #include "RenderThread.h"
10 #include "ui_viewer.h"
11
12 class Viewer : public QMainWindow
13 {
14     Q_OBJECT
15
16 public:
17     Viewer(Renderer* b, QWidget *parent = 0)
18         : QMainWindow(parent), renderer(b), image(
19             b->GetBitmap()->
20                 GetWidth(),
21             b->GetBitmap()->
22                 GetHeight(), QImage
23                 ::Format_RGB888)
24     {
25         ui.setupUi(this);
26         thread = new RenderThread(this, renderer, &image);
27         transformedImage = image;
28         // connect(thread,
29         //           SIGNAL(PixelAdded(int, int, int, int)),
30         //           this, SLOT(onPixelAdded(int, int, int, int)));
31         resize(image.width() + 25, image.height() + 50 + this->ui.
32                 mainToolBar->size().height());
33         setMouseTracking(true);
34     }
35     ~Viewer();
36
37     QImage image;
38     QImage transformedImage;
39     RenderThread* thread;
40     float zoom = 1;
41     float panX = 0;
42     float panY = 0;
43     float previousX = 0;
44     float previousY = 0;
45
46 private:
47     Ui::ViewerClass ui;
48     Renderer* renderer;
49
50 protected:
```

```

46 void paintEvent(QPaintEvent *e);
47 void wheelEvent(QWheelEvent *event)
48 {
49
50     zoom += (event->delta() / abs(event->delta())) * 0.1 ;
51     if(zoom < 0.1)
52         zoom = 0.1;
53     else if(zoom > 1000.0f)
54         zoom = 1000.0f;
55     qDebug() << zoom;
56 //     QMatrix trans;
57 //     trans = trans.scale(zoom,zoom);
58 //     transformedImage = image.transformed(trans);
59
60     event->accept();
61     update();
62 }
63
64 void mouseMoveEvent(QMouseEvent* event)
65 {
66     if((event->buttons() & Qt::LeftButton) && QApplication::
67         keyboardModifiers() & Qt::AltModifier){
68         panX += event->pos().x() - previousX;
69         panY += event->pos().y() - previousY;
70     }
71
72     previousX = event->pos().x();
73     previousY = event->pos().y();
74
75     update();
76 }
77
78 void mousePressEvent(QMouseEvent* event)
79 {
80     previousX = event->pos().x();
81     previousY = event->pos().y();
82 }
83
84 private slots:
85     void on_actionRender_triggered();
86     void on_actionSave_triggered();
87     void on_actionStopRender_triggered();
88     void on_actionLoadScene_triggered();
89 };
90
91 #endif // VIEWER_H

```

Listing 4.7: viewer source file

```
1 #include "viewer.h"
2 #include <QMessageBox>
3 #include "XMLParser.h"
4
5
6 void Viewer::paintEvent(QPaintEvent *e)
7 {
8     QPainter painter(this);
9
10    painter.setBrush(QColor(170, 170, 170));
11    painter.drawRect(0, 0, this->geometry().size().width(), this->
12                     geometry().size().height());
13    QMatrix trans;
14    trans = trans.scale(zoom,zoom);
15    transformedImage = image.copy(0, 0, renderer->GetBitmap()->GetWidth
16                                 (), renderer->GetBitmap()->GetHeight()).transformed(trans);
17    QRect rect(transformedImage.rect());
18    QRect devRect(panX, panY, painter.device()->width(), painter.device
19                  ()->height() + ui.mainToolBar->height());
20    rect.moveCenter(devRect.center());
21    painter.drawImage(rect.topLeft(), transformedImage);
22    painter.setBrush(Qt::transparent);
23    painter.setPen(QPen(QColor(0, 0, 0), 1));
24    QPainterPath path;
25    path.addRect(rect);
26    painter.fillPath(path, Qt::transparent);
27    painter.drawPath(path);
28 }
29
30 void Viewer::on_actionRender_triggered()
31 {
32     if (renderer == nullptr)
33     {
34         QMessageBox::warning(this, tr("Scene File"),
35                             tr("No scene file opened."));
36         return;
37     }
38     image.fill(QColor(Qt::black).rgb());
39     update();
40     thread->start();
41 }
42
43 }
44
45
46 void Viewer::on_actionSave_triggered()
47 {
48     QStringList mimeTypeFilters;
```

```

49     foreach (const QByteArray &MimeTypeName, QImageReader::
50             supportedMimeTypes())
51         mimeTypeFilters.append(MimeTypeName);
52     mimeTypeFilters.sort();
53     const QStringList picturesLocations = QStandardPaths::
54         standardLocations(QStandardPaths::PicturesLocation);
55     QDialog dialog(this, tr("Save Image"),
56                   picturesLocations.isEmpty() ? QDir::currentPath
57                   () : picturesLocations.first());
58     dialog.setAcceptMode(QFileDialog::AcceptSave);
59     dialog.setMimeTypeFilters(mimeTypeFilters);
60     dialog.selectMimeTypeFilter("image/ ppm");
61     if(dialog.exec())
62     {
63         if(!dialog.selectedFiles().isEmpty())
64         {
65             QRect rect(0, 0, renderer->GetBitmap()->GetWidth(),
66                         renderer->GetBitmap()->GetHeight());
67             QImage croppedImage = image.copy(rect);
68             QImageWriter writer(dialog.selectedFiles().at(0));
69             writer.write(croppedImage);
70         }
71     }
72 }
73
74 void Viewer::on_actionStopRender_triggered()
75 {
76     renderer->OnStop();
77 }
78
79 void Viewer::on_actionLoadScene_triggered()
80 {
81     QString fileName =
82         QFileDialog::getOpenFileName(this, tr("Open Scene File"),
83                                     QDir::currentPath(),
84                                     tr("XML Files (*.xml)"));
85
86     if (fileName.isEmpty())
87         return;
88
89     if (!fileName.endsWith("xml", Qt::CaseInsensitive))
90     {
91         QMessageBox::warning(this, tr("Scene File"),
92                             tr("Cannot read file %1:\n%2.")
93                             .arg(fileName)
94                             .arg("Incorrect file type"));
95         return;
96     }
97
98     QFile file(fileName);
99     if (!file.open(QFile::ReadOnly | QFile::Text)) {
100         QMessageBox::warning(this, tr("Scene File"),
101                             tr("Scene File"));
102     }

```

```
97         tr("Cannot read file %1:\n%2.")
98         .arg(fileName)
99         .arg(file.errorString()));
100    return;
101 }
102
103 XMLParser parser;
104 renderer = parser.Read(&file);
105 if (renderer == nullptr) {
106     QMessageBox::warning(this, tr("Scene File"),
107                         tr("Parse error in file %1")
108                         .arg(fileName));
109 }
110 else {
111     thread->renderer = renderer;
112     auto handler = &RenderThread::UpdateScreen;
113     auto func = std::bind(handler, thread, std::placeholders::_1,
114                           std::placeholders::_2, std::placeholders::_3, std::
115                           placeholders::_4, std::placeholders::_5);
116     auto start_handler = &RenderThread::StartScreen;
117     auto start_func = std::bind(start_handler, thread, std::
118                                 placeholders::_1, std::placeholders::_2, std::placeholders
119                                 ::_3, std::placeholders::_4);
120     renderer->SetOnUpdateCallback(func);
121     renderer->SetOnStartCallback(start_func);
122 }
```

4.0.6 XMLParser

Listing 4.8: XMLParser header file

```
1 #pragma once
2
3 #include <QXmlStreamReader>
4 #include <QDebug>
5 #include <QtWidgets/QFileDialog>
6 #include <QString>
7 #include "Global.h"
8 #include "../PhotonMapping/Engine/Materials/EmissiveMaterial.h"
9 #include "../PhotonMapping/Engine/Integrators/PathTracingIntegrator.h"
10 #include "../PhotonMapping/Engine/Light/DiffuseAreaLight.h"
11 #include "../PhotonMapping/Engine/Integrators/PhotonMappingIntegrator.h"
12
13 #include "../PhotonMapping/Engine/Materials/BlinnMaterial.h"
14 #include "../PhotonMapping/Engine/Materials/RoughDielectricMaterial.h"
15 #include "../PhotonMapping/Engine/Materials/WardMaterial.h"
16 #include "../PhotonMapping/Engine/Materials/TwoSidedMaterial.h"
17 #include "../PhotonMapping/Engine/Materials/BlendMaterial.h"
18 #include "../PhotonMapping/Engine/Volume/ConstantVolume.h"
19 #include "../PhotonMapping/Engine/Volume/RayMarcher.h"
20 #include "../PhotonMapping/Engine/Volume/VoxelGrid.h"
21 #include "../PhotonMapping/Engine/Light/IES.h"
22 #include "../PhotonMapping/Engine/Light/IESLight.h"
23 #include "../PhotonMapping/Engine/Light/DistantDirectionalLight.h"
24
25 struct RendererFactory
26 {
27     RendererFactory();
28     bool IsValid();
29     int width;
30     int height;
31     Camera* camera;
32     Scene* scene;
33     Integrator* integrator;
34     VolumeIntegrator* volumeIntegrator;
35     Volume* volume;
36     Sampler* sampler;
37     Bitmap* bitmap;
38     std::vector<Primitive*> primitives;
39     std::vector<Light*> lights;
40     std::vector<Primitive*> specularSurfaces;
41 };
42 inline RendererFactory::RendererFactory()
43 {
44     width = 640;
45     height = 480;
46     camera = nullptr;
47     scene = nullptr;
48     integrator = nullptr;
```

```

49     sampler = nullptr;
50     bitmap = nullptr;
51     volumeIntegrator = nullptr;
52     volume = nullptr;
53 }
54
55 inline bool RendererFactory::IsValid()
56 {
57     return !(camera == nullptr ||
58             scene == nullptr ||
59             integrator == nullptr ||
60             sampler == nullptr ||
61             bitmap == nullptr);
62 }
63
64 class XMLParser
65 {
66 public:
67     void ReadFloat3(const char* name, float& x, float& y, float& z)
68         const;
69     void ReadFloat(const char* name, float& x) const;
70     void ReadInteger(const char* name, int& x) const;
71     void XMLParser::ReadBoolean(const char* name, bool& x) const;
72     Matrix ParseCameraTransform();
73     Matrix ParseTransform();
74     Camera* ParseCamera();
75     Material* ParseDiffuseMaterial();
76     Material* ParseRefractionMaterial();
77     Material* ParseEmitterMaterial();
78     Material* ParseReflectiveMaterial();
79     Material* ParseBlinnMaterial();
80     Material* ParsePlasticMaterial();
81     Material* ParseRoughRefractiveMaterial();
82     Material* ParseWardMaterial();
83     Material* ParseTwoSided();
84     Material* ParseBlendMaterial();
85     Material* ParseMaterial();
86     Primitive* ParseObj();
87     Primitive* ParseQuadrilateral();
88     Primitive* ParseGeometry();
89     Texture* ParseTexture();
90     void ParsePathIntegrator();
91     void ParsePhotonIntegrator();
92     void ParseWhittedIntegrator();
93     void ParseIntegrator();
94     void ParseCommonSettings();
95     void ParseDomeLight();
96     void ParseIESLight();
97     void ParsePointLight();
98     void ParseSpotLight();
99     void ParseDistantDiscLight();
100    void ParseDirectionalLight();

```

```

100    void ParseLight();
101    void ParseAreaLight();
102    void ParseRandomSampler();
103    void ParseSampler();
104    PhaseFunction* ParsePhaseFunction();
105    Volume* ParseHomogeneousVolume();
106    Volume* ParseGridVolume();
107    void ParseVolumeIntegrator();
108    Renderer* Read(QIODevice* device);
109
110 private:
111     QXmlStreamReader xml;
112     RendererFactory renderer;
113
114 };
115
116
117
118 inline void XMLParser::ReadFloat3(const char* name, float& x, float& y,
119                                    float& z) const
120 {
121     auto attributes = xml.attributes();
122     QString l = attributes.value(name).toString();
123     QStringList numberStrings = l.split(' ', QString::SkipEmptyParts);
124     if (numberStrings.size() != 3)
125         qt_assert("Error parsing float3", "", xml.columnNumber());
126     x = numberStrings[0].toFloat();
127     y = numberStrings[1].toFloat();
128     z = numberStrings[2].toFloat();
129 }
130
131 inline void XMLParser::ReadFloat(const char* name, float& x) const
132 {
133     auto attributes = xml.attributes();
134     x = attributes.value(name).toFloat();
135 }
136
137 inline void XMLParser::ReadInteger(const char* name, int& x) const
138 {
139     auto attributes = xml.attributes();
140     x = attributes.value(name).toInt();
141 }
142
143 inline void XMLParser::ReadBoolean(const char* name, bool& x) const
144 {
145     auto attributes = xml.attributes();
146     auto boolean = attributes.value(name).toString().toLower();
147     if (boolean == "false")
148         x = false;
149     else if (boolean == "true")
150         x = true;

```

```

151
152 inline Matrix XMLParser::ParseCameraTransform()
153 {
154     Q_ASSERT(xml.isStartElement() && xml.name() == "transform");
155     xml.readNext();
156     while (!(xml.name() == "transform" && xml.isEndElement()))
157     {
158         if (xml.isStartElement())
159         {
160             if (xml.name() == "lookat")
161             {
162                 Point eye;
163                 Point center;
164                 Vector up;
165                 float x, y, z;
166                 auto attributes = xml.attributes();
167                 if (attributes.hasAttribute("target"))
168                 {
169                     ReadFloat3("target", x, y, z);
170                     qDebug() << '\t' << x << y << z;
171                     center = Point(x, y, z);
172                 }
173                 if (attributes.hasAttribute("origin"))
174                 {
175                     ReadFloat3("origin", x, y, z);
176                     qDebug() << '\t' << x << y << z;
177                     eye = Point(x, y, z);
178                 }
179                 if (attributes.hasAttribute("up"))
180                 {
181                     ReadFloat3("up", x, y, z);
182                     qDebug() << '\t' << x << y << z;
183                     up = Vector(x, y, z);
184                 }
185
186                 Matrix m = Matrix::LookAt(eye, center, up);
187                 Matrix::Inverse(m);
188                 return m;
189             }
190             if (xml.name() == "max")
191             {
192                 Vector translation;
193                 Vector rotation;
194                 auto attributes = xml.attributes();
195                 float x, y, z;
196                 if (attributes.hasAttribute("translation"))
197                 {
198                     ReadFloat3("translation", x, y, z);
199                     qDebug() << '\t' << x << y << z;
200                     translation = Vector(x, y, z);
201                 }
202                 if (attributes.hasAttribute("rotation"))

```

```

203
204         {
205             ReadFloat3("rotation", x, y, z);
206             qDebug() << '\t' << x << y << z;
207             rotation = Vector(x, y, z);
208         }
209         return XYZMax(translation, rotation);
210     }
211     if (xml.name() == "maya")
212     {
213         Vector translation;
214         Vector rotation;
215         auto attributes = xml.attributes();
216         float x, y, z;
217         if (attributes.hasAttribute("translation"))
218         {
219             ReadFloat3("translation", x, y, z);
220             qDebug() << '\t' << x << y << z;
221             translation = Vector(x, y, z);
222         }
223         if (attributes.hasAttribute("rotation"))
224         {
225             ReadFloat3("rotation", x, y, z);
226             qDebug() << '\t' << x << y << z;
227             rotation = Vector(x, y, z);
228         }
229         return XYZTransformMaya(translation, rotation);
230     }
231
232     }
233     xml.readNext();
234 }
235 return Matrix();
236 }
237
238 inline Matrix XMLParser::ParseTransform()
239 {
240     Q_ASSERT(xml.isStartElement() && xml.name() == "transform");
241     xml.readNext();
242     while (!(xml.name() == "transform" && xml.isEndElement()))
243     {
244         if (xml.isStartElement())
245         {
246             if (xml.name() == "max")
247             {
248                 Vector translation;
249                 Vector rotation;
250                 auto attributes = xml.attributes();
251                 float x, y, z;
252                 if (attributes.hasAttribute("translation"))
253                 {
254                     ReadFloat3("translation", x, y, z);

```

```

255         qDebug() << '\t' << x << y << z;
256         translation = Vector(x, y, z);
257     }
258     if (attributes.hasAttribute("rotation"))
259     {
260         ReadFloat3("rotation", x, y, z);
261         qDebug() << '\t' << x << y << z;
262         rotation = Vector(x, y, z);
263     }
264     return XYZTransformMax(translation, rotation);
265 }
266 if (xml.name() == "maya")
267 {
268     Vector translation;
269     Vector rotation;
270     auto attributes = xml.attributes();
271     float x, y, z;
272     if (attributes.hasAttribute("translation"))
273     {
274         ReadFloat3("translation", x, y, z);
275         qDebug() << '\t' << x << y << z;
276         translation = Vector(x, y, z);
277     }
278     if (attributes.hasAttribute("rotation"))
279     {
280         ReadFloat3("rotation", x, y, z);
281         qDebug() << '\t' << x << y << z;
282         rotation = Vector(x, y, z);
283     }
284     return XYZTransformMaya(translation, rotation);
285 }
286 }
287     }
288     xml.readNext();
289 }
290     return Matrix();
291 }
292
293 inline Camera * XMLParser::ParseCamera()
294 {
295     Q_ASSERT(xml.isStartElement() && xml.name() == "camera");
296     QString type = xml.attributes().value("type").toString();
297     qDebug() << type << "->";
298     float fov;
299     Matrix m;
300     QXmlStreamReader::TokenType token = xml.readNext();
301     while (token != QXmlStreamReader::EndElement && xml.name() != "
302         camera")
303     {
304         xml.readNext();
305         if (xml.isStartElement())
306         {

```

```

306         qDebug() << "\t" << xml.name();
307     if (xml.name() == "transform")
308     {
309         m = ParseCameraTransform();
310     }
311     if (xml.name() == "fov")
312     {
313         ReadFloat("value", fov);
314         qDebug() << '\t' << fov;
315     }
316 }
317 if (xml.hasError())
318 {
319     return nullptr;
320 }
321 return new PinholeCamera(m, fov, renderer.width / float(renderer.
322 height), 0.1, 1000.0f);
323 }
324
325 inline Material* XMLParser::ParseDiffuseMaterial()
326 {
327     xml.readNext();
328     float x=1.0f, y=1.0f, z=1.0f;
329     while (!(xml.name().contains("material", Qt::CaseInsensitive) &&
330             xml.isEndElement()))
331     {
332         if (xml.isStartElement())
333         {
334             if (xml.name() == "albedo")
335             {
336                 auto attributes = xml.attributes();
337                 if (attributes.hasAttribute("value"))
338                 {
339                     ReadFloat3("value", x, y, z);
340                     return new DiffuseMaterial(new SolidColourTexture(x
341                         , y, z), Colour(x, y, z));
342                 }
343                 if (attributes.hasAttribute("bitmap"))
344                 {
345                     auto filePath = attributes.value("bitmap").toString
346                         ();
347                     auto tileU = attributes.hasAttribute("tileU") ?
348                         attributes.value("tileU").toFloat() : 1.0f;
349                     auto tileV = attributes.hasAttribute("tileV") ?
350                         attributes.value("tileV").toFloat() : 1.0f;
351                     auto offsetU = attributes.hasAttribute("offsetU") ?
352                         attributes.value("offsetU").toFloat() : 0.0f;
353                     auto offsetV = attributes.hasAttribute("offsetV") ?
354                         attributes.value("offsetV").toFloat() : 0.0f;
355
356                     if (filePath.endsWith(".pfm"))

```

```

350             {
351                 return new DiffuseMaterial(new BitmapTexture(
352                     filePath.toStdString(), "a", tileU, tileV,
353                     offsetU, offsetV, true, 1), Colour(1, 1, 1)
354                     );
355             }
356         }
357     }
358 }
359 }
360 }
361     xml.readNext();
362 }
363 }
364     return new DiffuseMaterial(new SolidColourTexture(x, y, z), Colour(
365         x, y, z));
366 }
367 inline Material* XMLParser::ParseRefractionMaterial()
368 {
369     xml.readNext();
370     Colour ks(1, 1, 1);
371     float ior = 1.5;
372     while (!(xml.name() == "material" && xml.isEndElement()))
373     {
374         if (xml.isStartElement())
375         {
376             if (xml.name() == "colour")
377             {
378                 float x, y, z;
379                 ReadFloat3("value", x, y, z);
380                 ks = Colour(x, y, z);
381             }
382             if (xml.name() == "ior")
383             {
384                 ReadFloat("value", ior);
385             }
386         }
387         xml.readNext();
388     }
389     return new DielectricMaterial(ks, ior);
390 }
391 }
392 inline Material* XMLParser::ParseEmitterMaterial()
393 {
394     xml.readNext();
395     float x=1.0f, y=1.0f, z=1.0f;

```

```

396     float multiplier = 1.0f;
397     Texture* albedo = nullptr;
398     while (!(xml.name() == "material" && xml.isEndElement()))
399     {
400         if (xml.isStartElement())
401         {
402             if (xml.name() == "multiplier")
403             {
404                 ReadFloat("value", multiplier);
405             }
406             if (xml.name() == "albedo")
407             {
408                 auto attributes = xml.attributes();
409                 if (attributes.hasAttribute("value"))
410                 {
411                     ReadFloat3("value", x, y, z);
412                     albedo = new SolidColourTexture(x, y, z, 2.2);
413                 }
414                 if (attributes.hasAttribute("bitmap"))
415                 {
416                     auto filePath = attributes.value("bitmap").toString
417                         ();
418                     auto tileU = attributes.getAttribute("tileU") ?
419                         attributes.value("tileU").toFloat() : 1.0f;
420                     auto tileV = attributes.getAttribute("tileV") ?
421                         attributes.value("tileV").toFloat() : 1.0f;
422                     auto offsetU = attributes.getAttribute("offsetU") ?
423                         attributes.value("offsetU").toFloat() : 0.0f;
424                     auto offsetV = attributes.getAttribute("offsetV") ?
425                         attributes.value("offsetV").toFloat() : 0.0f;
426                     if (filePath.endsWith(".pfm"))
427                     {
428                         albedo = new BitmapTexture(filePath.toStdString
429                             (), "a", tileU, tileV, offsetU, offsetV,
430                             true, 1);
431                     }
432                 }
433             }
434             xml.readNext();
435         }
436         if (albedo == nullptr)
437             return new EmissiveMaterial(new SolidColourTexture(x, y, z),
438                 multiplier);
439     }

```

```

439         return new EmissiveMaterial(albedo, multiplier);
440     }
441
442     inline Material* XMLParser::ParseReflectiveMaterial()
443     {
444         xml.readNext();
445         Colour ks(1, 1, 1);
446         float ior = 1.5f;
447         float k = 50.0f;
448         while (!(xml.name().contains("material", Qt::CaseInsensitive) && xml
449             .isEndElement())))
450         {
451             if (xml.isStartElement())
452             {
453                 if (xml.name() == "colour")
454                 {
455                     float x, y, z;
456                     ReadFloat3("value", x, y, z);
457                     ks = Colour(x, y, z);
458                 }
459                 if (xml.name() == "ior")
460                 {
461                     ReadFloat("value", ior);
462                 }
463                 if (xml.name() == "k")
464                 {
465                     ReadFloat("value", k);
466                 }
467             }
468             xml.readNext();
469         }
470         return new SpecularMaterial(ks, ior, k);
471     }
472
473     inline Material* XMLParser::ParseBlinnMaterial()
474     {
475         xml.readNext();
476         Texture* diffuseTexture = new SolidColourTexture(1, 1, 1, 2.2);
477         float roughness = 1000.0f;
478         float ior = 1.5f;
479         while (!(xml.name() == "material" && xml.isEndElement()))
480         {
481             if (xml.isStartElement())
482             {
483                 if (xml.name() == "albedo")
484                 {
485                     auto attributes = xml.attributes();
486                     if (attributes.hasAttribute("value"))
487                     {
488                         float x, y, z;
489                         ReadFloat3("value", x, y, z);

```



```

532     }
533     return new BlinnMaterial(roughness, ior, diffuseTexture);
534 }
535
536 inline Material* XMLParser::ParsePlasticMaterial()
537 {
538     xml.readNext();
539     Texture* diffuseTexture = new SolidColourTexture(1, 1, 1, 2.2);
540     Texture* roughnessTexture = nullptr;
541     Texture* specularColorTexture = nullptr;
542     Texture* normalTexture = nullptr;
543     Colour kd(1.0f);
544     Colour ks(1.0f);
545     float roughness = 1000;
546     float ior = 1.5f;
547     float multiplier = 1.0f;
548     while (!(xml.name() == "material" && xml.isEndElement()))
549     {
550         if (xml.isStartElement())
551         {
552             if (xml.name() == "albedo")
553             {
554                 auto attributes = xml.attributes();
555                 if (attributes.hasAttribute("value"))
556                 {
557                     float x, y, z;
558                     ReadFloat3("value", x, y, z);
559                     diffuseTexture = new SolidColourTexture(x, y, z,
560                                                 2.2);
561                 }
562                 if (attributes.hasAttribute("bitmap"))
563                 {
564                     auto filePath = attributes.value("bitmap").toString
565                                     ();
566                     auto tileU = attributes.getAttribute("tileU") ?
567                         attributes.value("tileU").toFloat() : 1.0f;
568                     auto tileV = attributes.getAttribute("tileV") ?
569                         attributes.value("tileV").toFloat() : 1.0f;
570                     auto offsetU = attributes.getAttribute("offsetU") ?
571                         attributes.value("offsetU").toFloat() : 0.0f;
572                     auto offsetV = attributes.getAttribute("offsetV") ?
573                         attributes.value("offsetV").toFloat() : 0.0f;
574
575                     if (filePath.endsWith(".pfm"))
576                     {
577                         diffuseTexture = new BitmapTexture(filePath.
578                                         toStdString(), "a", tileU, tileV, offsetU,
579                                         offsetV, true, 1);
580                     }
581                     if (filePath.endsWith(".ppm"))
582                     {
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
```

```

575                     diffuseTexture = new BitmapTexture(filePath.
576                                         toStdString(), tileU, tileV, offsetU,
577                                         offsetV, true);
578                 }
579             }
580         if (xml.name() == "specular")
581     {
582         auto attributes = xml.attributes();
583         if (attributes.hasAttribute("value"))
584     {
585         float x, y, z;
586         ReadFloat3("value", x, y, z);
587         specularColorTexture = new SolidColourTexture(x, y,
588                                         z, 2.2);
589     }
590     if (attributes.hasAttribute("bitmap"))
591     {
592         auto filePath = attributes.value("bitmap").toString
593             ();
594         auto tileU = attributes.getAttribute("tileU") ?
595             attributes.value("tileU").toFloat() : 1.0f;
596         auto tileV = attributes.getAttribute("tileV") ?
597             attributes.value("tileV").toFloat() : 1.0f;
598         auto offsetU = attributes.getAttribute("offsetU") ?
599             attributes.value("offsetU").toFloat() : 0.0f;
600         auto offsetV = attributes.getAttribute("offsetV") ?
601             attributes.value("offsetV").toFloat() : 0.0f;
602
603         if (filePath.endsWith(".pfm"))
604     {
605         specularColorTexture = new BitmapTexture(
606             filePath.toStdString(), "a", tileU, tileV,
607             offsetU, offsetV, true, 1);
608     }
609     if (filePath.endsWith(".ppm"))
610     {
611         specularColorTexture = new BitmapTexture(
612             filePath.toStdString(), tileU, tileV,
613             offsetU, offsetV, true);
614     }
615 }
616
617 if (xml.name() == "normal")
618 {
619     auto attributes = xml.attributes();
620     if (attributes.hasAttribute("multiplier"))
621     {
622         ReadFloat("multiplier", multiplier);
623     }
624     if (attributes.hasAttribute("bitmap"))

```

```

615
616    {
617        auto filePath = attributes.value("bitmap").toString()
618        ();
619        auto tileU = attributes.hasAttribute("tileU") ?
620            attributes.value("tileU").toFloat() : 1.0f;
621        auto tileV = attributes.hasAttribute("tileV") ?
622            attributes.value("tileV").toFloat() : 1.0f;
623        auto offsetU = attributes.hasAttribute("offsetU") ?
624            attributes.value("offsetU").toFloat() : 0.0f;
625        auto offsetV = attributes.hasAttribute("offsetV") ?
626            attributes.value("offsetV").toFloat() : 0.0f;
627
628        if (filePath.endsWith(".pfm"))
629        {
630            normalTexture = new BitmapTexture(filePath.
631                toStdString(), "a", tileU, tileV, offsetU,
632                offsetV, true, 1);
633        }
634        if (filePath.endsWith(".ppm"))
635        {
636            normalTexture = new BitmapTexture(filePath.
637                toStdString(), tileU, tileV, offsetU,
638                offsetV, true, 1);
639        }
640    }
641    if (xml.name() == "roughness")
642    {
643        auto attributes = xml.attributes();
644        if (attributes.hasAttribute("value"))
645        {
646            float x;
647            ReadFloat("value", x);
648            roughness = x;
649        }
650        if (attributes.hasAttribute("bitmap"))
651        {
652            auto filePath = attributes.value("bitmap").toString()
653            ();
654            auto tileU = attributes.hasAttribute("tileU") ?
655                attributes.value("tileU").toFloat() : 1.0f;
656            auto tileV = attributes.hasAttribute("tileV") ?
657                attributes.value("tileV").toFloat() : 1.0f;
658            auto offsetU = attributes.hasAttribute("offsetU") ?
659                attributes.value("offsetU").toFloat() : 0.0f;
660            auto offsetV = attributes.hasAttribute("offsetV") ?
661                attributes.value("offsetV").toFloat() : 0.0f;
662
663            if (filePath.endsWith(".pfm"))
664            {
665                roughnessTexture = new BitmapTexture(filePath.
666                    toStdString(), "a", tileU, tileV, offsetU,

```

```

                                offsetV, true, 1);
652    }
653    if (filePath.endsWith(".ppm"))
654    {
655        roughnessTexture = new BitmapTexture(filePath.
656                                              toStdString(), tileU, tileV, offsetU,
657                                              offsetV, true, 1);
658    }
659    if (xml.name() == "ior")
660    {
661        auto attributes = xml.attributes();
662        if (attributes.hasAttribute("value"))
663        {
664            float x;
665            ReadFloat("value", x);
666            ior = x;
667        }
668    }
669    xml.readNext();
670}
671 return new PlasticMaterial1(kd, ks, roughness, ior, diffuseTexture,
672                             roughnessTexture, specularColorTexture, normalTexture,
673                             multiplier);
674}
675 inline Material* XMLParser::ParseRoughRefractiveMaterial()
676{
677    xml.readNext();
678    Colour kd = Colour(1.0f, 1.0f, 1.0f) ^ (1 / 2.2);
679    Texture* glossinessTexture = nullptr;
680    Texture* normalTexture = nullptr;
681    float roughness = 1;
682    float ior = 1.5f;
683    float multiplier = 1.0f;
684    while (!(xml.name() == "material" && xml.isEndElement()))
685    {
686        if (xml.isStartElement())
687        {
688            if (xml.name() == "glossiness")
689            {
690                auto attributes = xml.attributes();
691                if (attributes.hasAttribute("value"))
692                {
693                    float x, y, z;
694                    ReadFloat3("value", x, y, z);
695                    glossinessTexture = new SolidColourTexture(x, y, z,
696                                                    2.2);
697                }
698                if (attributes.hasAttribute("bitmap"))

```

```

698
699     {
700         auto filePath = attributes.value("bitmap").toString()
701             ();
702         auto tileU = attributes.hasAttribute("tileU") ?
703             attributes.value("tileU").toFloat() : 1.0f;
704         auto tileV = attributes.hasAttribute("tileV") ?
705             attributes.value("tileV").toFloat() : 1.0f;
706         auto offsetU = attributes.hasAttribute("offsetU") ?
707             attributes.value("offsetU").toFloat() : 0.0f;
708         auto offsetV = attributes.hasAttribute("offsetV") ?
709             attributes.value("offsetV").toFloat() : 0.0f;
710
711         if (filePath.endsWith(".pfm"))
712         {
713             glossinessTexture = new BitmapTexture(filePath.
714                 toStdString(), "a", tileU, tileV, offsetU,
715                 offsetV, true, 1);
716         }
717         if (filePath.endsWith(".ppm"))
718         {
719             glossinessTexture = new BitmapTexture(filePath.
720                 toStdString(), tileU, tileV, offsetU,
721                 offsetV, true);
722         }
723     }
724     if (xml.name() == "normal")
725     {
726         auto attributes = xml.attributes();
727         if (attributes.hasAttribute("multiplier"))
728         {
729             ReadFloat("multiplier", multiplier);
730         }
731         if (attributes.hasAttribute("bitmap"))
732         {
733             auto filePath = attributes.value("bitmap").toString()
734                 ();
735             auto tileU = attributes.hasAttribute("tileU") ?
736                 attributes.value("tileU").toFloat() : 1.0f;
737             auto tileV = attributes.hasAttribute("tileV") ?
738                 attributes.value("tileV").toFloat() : 1.0f;
739             auto offsetU = attributes.hasAttribute("offsetU") ?
740                 attributes.value("offsetU").toFloat() : 0.0f;
741             auto offsetV = attributes.hasAttribute("offsetV") ?
742                 attributes.value("offsetV").toFloat() : 0.0f;
743
744             if (filePath.endsWith(".pfm"))
745             {
746                 normalTexture = new BitmapTexture(filePath.
747                     toStdString(), "a", tileU, tileV, offsetU,
748                     offsetV, true, 1);
749             }
750         }
751     }

```

```

734         if (filePath.endsWith(".ppm"))
735     {
736         normalTexture = new BitmapTexture(filePath.
737                                         toStdString(), tileU, tileV, offsetU,
738                                         offsetV, true, 1);
739     }
740     if (xml.name() == "roughness")
741     {
742         auto attributes = xml.attributes();
743         if (attributes.hasAttribute("value"))
744         {
745             float x;
746             ReadFloat("value", x);
747             roughness = x;
748         }
749     }
750     if (xml.name() == "ior")
751     {
752         auto attributes = xml.attributes();
753         if (attributes.hasAttribute("value"))
754         {
755             float x;
756             ReadFloat("value", x);
757             ior = x;
758         }
759     }
760     xml.readNext();
761 }
762 return new RoughDielectricMaterial(kd, ior, roughness,
763                                     glossinessTexture);
764 }
765 }
766
767 inline Material* XMLParser::ParseWardMaterial()
768 {
769     xml.readNext();
770     Texture* anisotropyMap = nullptr;
771     Colour ks(1.0f);
772     float ior = 1.5f;
773     float ax = 0.5;
774     float ay = 0.5;
775     while (!(xml.name().contains("material", Qt::CaseInsensitive) &&
776             xml.isEndElement())))
777     {
778         if (xml.isStartElement())
779         {
780             auto attributes = xml.attributes();
781             if (xml.name() == "specular")
782             {

```

```

782     if (attributes.hasAttribute("value"))
783     {
784         float x, y, z;
785         ReadFloat3("value", x, y, z);
786         ks = Colour(x, y, z) ^ (1 / 2.2);
787     }
788 }
789 if (xml.name() == "ax")
790 {
791     if (attributes.hasAttribute("value"))
792         ReadFloat("value", ax);
793 }
794 if (xml.name() == "ay")
795 {
796     if (attributes.hasAttribute("value"))
797         ReadFloat("value", ay);
798 }
799 if (xml.name() == "ior")
800 {
801     if (attributes.hasAttribute("value"))
802     {
803         ReadFloat("value", ior);
804     }
805 }
806 if (xml.name() == "anisotropy")
807 {
808     if (attributes.hasAttribute("bitmap"))
809     {
810         auto filePath = attributes.value("bitmap").toString
811             ();
812         auto tileU = attributes.getAttribute("tileU") ?
813             attributes.value("tileU").toFloat() : 1.0f;
814         auto tileV = attributes.getAttribute("tileV") ?
815             attributes.value("tileV").toFloat() : 1.0f;
816         auto offsetU = attributes.getAttribute("offsetU") ?
817             attributes.value("offsetU").toFloat() : 0.0f;
818         auto offsetV = attributes.getAttribute("offsetV") ?
819             attributes.value("offsetV").toFloat() : 0.0f;
820
821         if (filePath.endsWith(".pfm"))
822         {
823             anisotropyMap = new BitmapTexture(filePath.
824                 toStdString(), "a", tileU, tileV, offsetU,
825                 offsetV, true, 1);
826         }
827         if (filePath.endsWith(".ppm"))
828         {
829             anisotropyMap = new BitmapTexture(filePath.
830                 toStdString(), tileU, tileV, offsetU,
831                 offsetV, true, 1);
832         }
833     }
834 }

```

```

825             }
826         }
827     }
828     xml.readNext();
829 }
830     return new WardMaterial(ks, ax, ay, anisotropyMap);
831 }
832
833 inline Material* XMLParser::ParseTwoSided()
834 {
835
836     xml.readNext();
837     Material* frontMaterial = nullptr;
838     Material* backMaterial = nullptr;
839     Texture* blendTexture = nullptr;
840     float blendAmount = 1.0f;
841     bool isTransluscent = false;
842     bool flipDirection = false;
843
844     while (!(xml.name() == "material" && xml.isEndElement()))
845     {
846         if (xml.isStartElement())
847         {
848             auto attributes = xml.attributes();
849             if (xml.name() == "frontMaterial")
850             {
851                 frontMaterial = ParseMaterial();
852             }
853             if (xml.name() == "backMaterial")
854             {
855                 backMaterial = ParseMaterial();
856             }
857             if (xml.name() == "blendAmount")
858             {
859                 if (attributes.hasAttribute("value"))
860                     ReadFloat("value", blendAmount);
861             }
862             if (xml.name() == "isTransluscent")
863             {
864                 if (attributes.hasAttribute("value"))
865                 {
866                     ReadBoolean("value", isTransluscent);
867                 }
868             }
869             if (xml.name() == "flipDirection")
870             {
871                 if (attributes.hasAttribute("value"))
872                 {
873                     ReadBoolean("value", flipDirection);
874                 }
875             }
876             if (xml.name() == "blendMap")

```

```

877         {
878             if (attributes.hasAttribute("bitmap"))
879             {
880                 auto filePath = attributes.value("bitmap").toString
881                             ();
882                 auto tileU = attributes.getAttribute("tileU") ?
883                             attributes.value("tileU").toFloat() : 1.0f;
884                 auto tileV = attributes.getAttribute("tileV") ?
885                             attributes.value("tileV").toFloat() : 1.0f;
886                 auto offsetU = attributes.getAttribute("offsetU") ?
887                             attributes.value("offsetU").toFloat() : 0.0f;
888                 auto offsetV = attributes.getAttribute("offsetV") ?
889                             attributes.value("offsetV").toFloat() : 0.0f;
890
891                 if (filePath.endsWith(".pfm"))
892                 {
893                     blendTexture = new BitmapTexture(filePath.
894                                         toStdString(), "a", tileU, tileV, offsetU,
895                                         offsetV, true, 1);
896                 }
897                 if (filePath.endsWith(".ppm"))
898                 {
899                     blendTexture = new BitmapTexture(filePath.
900                                         toStdString(), tileU, tileV, offsetU,
901                                         offsetV, true, 1);
902                 }
903             }
904             xml.readNext();
905         }
906         return new TwoSidedMaterial(frontMaterial, backMaterial,
907             blendTexture, blendAmount, isTransluscent, flipDirection);
908     }
909
910     inline Material* XMLParser::ParseBlendMaterial()
911     {
912         xml.readNext();
913         Material* baseMaterial = nullptr;
914         Material* coatMaterial = nullptr;
915         Texture* blendTexture = nullptr;
916         float blendAmount = 1.0f;
917
918         while (!(xml.name() == "material" && xml.isEndElement()))
919         {
920             if (xml.isStartElement())
921             {
922                 auto attributes = xml.attributes();
923                 if (xml.name() == "baseMaterial")
924                 {
925                     baseMaterial = ParseMaterial();
926                 }

```

```

919         if (xml.name() == "coatMaterial")
920     {
921         coatMaterial = ParseMaterial();
922     }
923     if (xml.name() == "blendAmount")
924     {
925         if (attributes.hasAttribute("value"))
926             ReadFloat("value", blendAmount);
927     }
928     if (xml.name() == "blendMap")
929     {
930         if (attributes.hasAttribute("bitmap"))
931     {
932             auto filePath = attributes.value("bitmap").toString
933             ();
934             auto tileU = attributes.hasAttribute("tileU") ?
935                 attributes.value("tileU").toFloat() : 1.0f;
936             auto tileV = attributes.hasAttribute("tileV") ?
937                 attributes.value("tileV").toFloat() : 1.0f;
938             auto offsetU = attributes.hasAttribute("offsetU") ?
939                 attributes.value("offsetU").toFloat() : 0.0f;
940             auto offsetV = attributes.hasAttribute("offsetV") ?
941                 attributes.value("offsetV").toFloat() : 0.0f;
942
943             if (filePath.endsWith(".pfm"))
944             {
945                 blendTexture = new BitmapTexture(filePath.
946                     toStdString(), "a", tileU, tileV, offsetU,
947                     offsetV, true, 1);
948             }
949             xml.readNext();
950         }
951         return new BlendMaterial(baseMaterial, coatMaterial, blendTexture,
952             blendAmount);
953     }
954     inline Material* XMLParser::ParseMaterial()
955     {
956         QString type = xml.attributes().value("type").toString();
957         qDebug() << type << "->";
958         xml.readNext();
959
960         if (type == "diffuse")

```

```

961     {
962         return ParseDiffuseMaterial();
963     }
964     if (type == "emissive")
965     {
966         return ParseEmitterMaterial();
967     }
968     if (type == "refractive")
969     {
970         return ParseRefractionMaterial();
971     }
972     if (type == "reflective")
973     {
974         return ParseReflectiveMaterial();
975     }
976     if (type == "blinn")
977     {
978         return ParseBlinnMaterial();
979     }
980     if (type == "plastic")
981     {
982         return ParsePlasticMaterial();
983     }
984     if (type == "roughRefractive")
985     {
986         return ParseRoughRefractiveMaterial();
987     }
988     if (type == "ward")
989     {
990         return ParseWardMaterial();
991     }
992     if (type == "twosided")
993     {
994         return ParseTwoSided();
995     }
996     if (type == "blend")
997     {
998         return ParseBlendMaterial();
999     }
1000     return nullptr;
1001 }
1002
1003 inline Primitive* XMLParser::ParseObj()
1004 {
1005     // get type of geometry
1006     QString type = xml.attributes().value("type").toString();
1007     qDebug() << "Geometry type " << type;
1008
1009     QXmlStreamReader::TokenType token = xml.readNext();
1010     Geometry* geometry = nullptr;
1011     Material* material = nullptr;
1012     Texture* alphaMap = nullptr;

```

```

1013     Matrix transformMatrix;
1014     QString filePath;
1015     xml.readNext();
1016     while (!(xml.name() == "geometry" && xml.isEndElement()))
1017     {
1018         if (xml.isStartElement())
1019         {
1020             auto attributes = xml.attributes();
1021             qDebug() << xml.name() << "->";
1022             if (xml.name() == "file")
1023             {
1024                 filePath = xml.attributes().value("value").toString();
1025                 qDebug() << '\t' << filePath;
1026             }
1027
1028             if (xml.name() == "alphaMap")
1029             {
1030                 if (attributes.hasAttribute("bitmap"))
1031                 {
1032                     auto path = attributes.value("bitmap").toString();
1033                     auto tileU = attributes.hasAttribute("tileU") ?
1034                         attributes.value("tileU").toFloat() : 1.0f;
1035                     auto tileV = attributes.hasAttribute("tileV") ?
1036                         attributes.value("tileV").toFloat() : 1.0f;
1037                     auto offsetU = attributes.hasAttribute("offsetU") ?
1038                         attributes.value("offsetU").toFloat() : 0.0f;
1039                     auto offsetV = attributes.hasAttribute("offsetV") ?
1040                         attributes.value("offsetV").toFloat() : 0.0f;
1041
1042                     if (path.endsWith(".pfm"))
1043                     {
1044                         alphaMap = new BitmapTexture(path.toStdString()
1045                             , "a", tileU, tileV, offsetU, offsetV, true
1046                             , 1);
1047                     }
1048
1049                     if (xml.name() == "transform")
1050                     {
1051                         qDebug() << "here";
1052                         transformMatrix = ParseTransform();
1053                     }
1054
1055                     if (xml.name() == "material")
1056                     {

```

```

1058             material = ParseMaterial();
1059         }
1060     }
1061     xml.readNext();
1062 }
1063 if (!filePath.isEmpty())
1064 {
1065     std::vector<Vertex> indices;
1066     std::vector<Point> points;
1067     std::vector<Normal> normals;
1068     std::vector<Point> uvws;
1069     LoadOBJ1(filePath.toStdString(), indices, points, normals, uvws
1070             );
1071     Transform* transform2 = new Transform(transformMatrix);
1072     Transform* invTransform2 = new Transform(transform2->Inverse())
1073             ;
1074     geometry = new TriangleMesh(transform2, invTransform2, indices,
1075             points, normals, uvws, alphaMap);
1076 }
1077 Primitive* monolithPrimitive = new GeometricPrimitive(geometry,
1078             material);
1079 auto accelerated = new BVH(monolithPrimitive);
1080 renderer.primitives.push_back(accelerated);
1081 if ((material->GetType() & BRDF::Type::REFRACTION) == BRDF::Type::
1082             REFRACTION)
1083 {
1084     renderer.specularSurfaces.push_back(accelerated);
1085 }
1086 return monolithPrimitive;
1087 }
1088 inline Primitive* XMLParser::ParseQuadrilateral()
1089 {
1090     QString type = xml.attributes().value("type").toString();
1091     qDebug() << "Geometry type " << type;
1092     QXmlStreamReader::TokenType token = xml.readNext();
1093     Geometry* geometry = nullptr;
1094     Material* material = nullptr;
1095     float width = 0;
1096     float height = 0;
1097     Matrix transformMatrix;
1098     while (token != QXmlStreamReader::EndElement && xml.name() != "
1099             geometry")
1100     {
1101         if (xml.isStartElement())
1102         {
1103             qDebug() << xml.name() << " -> ";
1104             if (xml.name() == "width")

```

```

1104         {
1105             width = xml.attributes().value("value").toFloat();
1106             qDebug() << width;
1107         }
1108         if (xml.name() == "height")
1109         {
1110             height = xml.attributes().value("value").toFloat();
1111             qDebug() << height;
1112         }
1113
1114         if (xml.name() == "transform")
1115         {
1116             transformMatrix = ParseTransform();
1117         }
1118
1119         if (xml.name() == "material")
1120         {
1121             material = ParseMaterial();
1122         }
1123
1124     }
1125     xml.readNextStartElement();
1126 }
1127 Transform* transform = new Transform(transformMatrix);
1128 geometry = new Quadrilateral(transform, new Transform(transform->
    Inverse()), width, height);
1129 Primitive* monolithPrimitive = new GeometricPrimitive(geometry,
    material);
1130 renderer.primitives.push_back(monolithPrimitive);
1131 return monolithPrimitive;
1132 }
1133
1134 inline Primitive* XMLParser::ParseGeometry()
1135 {
1136     Q_ASSERT(xml.isStartElement() && xml.name() == "geometry");
1137     QString type = xml.attributes().value("type").toString();
1138     if (type == "obj")
1139     {
1140         return ParseObj();
1141     }
1142     if (type == "quadrilateral")
1143     {
1144         return ParseQuadrilateral();
1145     }
1146
1147     return nullptr;
1148
1149 }
1150 }
1151
1152 inline Texture* XMLParser::ParseTexture()
1153 {

```

```

1154     return nullptr;
1155 }
1157
1158 inline void XMLParser::ParsePathIntegrator()
1159 {
1160     xml.readNext();
1161     int maxBounces = 6;
1162     while (!(xml.name() == "integrator" && xml.isEndElement()))
1163     {
1164         if (xml.isStartElement())
1165         {
1166             qDebug() << xml.name() << "->";
1167             if (xml.name() == "maxBounces")
1168             {
1169                 maxBounces = xml.attributes().value("value").toInt();
1170             }
1171         }
1172         xml.readNext();
1173     }
1174     renderer.integrator = new PathTracingIntegrator(maxBounces);
1175 }
1176
1177 inline void XMLParser::ParsePhotonIntegrator()
1178 {
1179     xml.readNext();
1180     int maxPhotons = 10000;
1181     int maxCaustics = 0;
1182     float maxRadius = 20;
1183
1184     while (!(xml.name() == "integrator" && xml.isEndElement()))
1185     {
1186         if (xml.isStartElement())
1187         {
1188             qDebug() << xml.name() << "->";
1189             if (xml.name() == "maxPhotons")
1190             {
1191                 maxPhotons = xml.attributes().value("value").toInt();
1192             }
1193             if (xml.name() == "maxRadius")
1194             {
1195                 maxRadius = xml.attributes().value("value").toInt();
1196             }
1197             if (xml.name() == "maxCausticPhotons")
1198             {
1199                 maxCaustics = xml.attributes().value("value").toInt();
1200                 qDebug() << "maxCausticPhotons: " << maxCaustics;
1201             }
1202         }
1203         xml.readNext();
1204     }

```

```

1205     PhotonMappingIntegrator* integrator = new PhotonMappingIntegrator(
1206         maxPhotons, maxCaustics, maxRadius);
1207     renderer.integrator = integrator;
1208 }
1209 inline void XMLParser::ParseWhittedIntegrator()
1210 {
1211     xml.readNext();
1212     int maxBounces = 10;
1213     while (!(xml.name() == "integrator" && xml.isEndElement()))
1214     {
1215         if (xml.isStartElement())
1216         {
1217             qDebug() << xml.name() << "->";
1218             if (xml.name() == "maxBounces")
1219             {
1220                 maxBounces = xml.attributes().value("value").toInt();
1221             }
1222         }
1223         xml.readNext();
1224     }
1225     renderer.integrator = new WhittedIntegrator(maxBounces);
1226 }
1227
1228 inline void XMLParser::ParseIntegrator()
1229 {
1230     Q_ASSERT(xml.isStartElement() && xml.name() == "integrator");
1231     QString type = xml.attributes().value("type").toString();
1232     qDebug() << type;
1233     if (type == "path")
1234     {
1235         ParsePathIntegrator();
1236     }
1237     if (type == "photon")
1238     {
1239         ParsePhotonIntegrator();
1240     }
1241     if (type == "whitted")
1242     {
1243         ParseWhittedIntegrator();
1244     }
1245 }
1246
1247 inline void XMLParser::ParseCommonSettings()
1248 {
1249     Q_ASSERT(xml.isStartElement() && xml.name() == "common");
1250     xml.readNext();
1251
1252     while (!(xml.name() == "common" && xml.isEndElement()))
1253     {
1254         if (xml.isStartElement())
1255         {

```

```

1256     qDebug() << xml.name() << " ->" ;
1257     if (xml.name() == "width")
1258     {
1259         renderer.width = xml.attributes().value("value").toInt
1260             ();
1261         qDebug() << '\t' << renderer.width;
1262     }
1263     if (xml.name() == "height")
1264     {
1265         renderer.height = xml.attributes().value("value").toInt
1266             ();
1267         qDebug() << '\t' << renderer.height;
1268     }
1269     xml.readNext();
1270 }
1271
1272 inline void XMLParser::ParseDomeLight()
1273 {
1274     xml.readNext();
1275     float multiplier = 1.0f;
1276     Texture* texture = nullptr;
1277     Transform transform;
1278     while (!(xml.name() == "light" && xml.isEndElement()))
1279     {
1280         if (xml.isStartElement())
1281         {
1282             qDebug() << "\t" << xml.name();
1283             if (xml.name() == "texture")
1284             {
1285                 auto attributes = xml.attributes();
1286                 float gamma = 1.0f;
1287                 float tileU = 1.0f;
1288                 float tileV = 1.0f;
1289                 float offsetU = 0.0f;
1290                 float offsetV = 0.0f;
1291                 if (attributes.hasAttribute("gamma"))
1292                 {
1293                     gamma = attributes.value("gamma").toFloat();
1294                 }
1295                 if (attributes.hasAttribute("pfm"))
1296                 {
1297                     auto filePath = attributes.value("pfm").toString();
1298                     texture = new BitmapTexture(filePath.toStdString(),
1299                         "pfm", tileU, tileV, offsetU, offsetV, true,
1300                         gamma);
1301                 }
1302                 if (attributes.hasAttribute("ppm"))
1303                 {
1304                     auto filePath = attributes.value("ppm").toString();

```

```

1303         texture = new BitmapTexture(filePath.toStdString(),
1304                                     tileU, tileV, offsetU, offsetV, true, gamma);
1305     }
1306     if (xml.name() == "multiplier")
1307     {
1308         ReadFloat("value", multiplier);
1309     }
1310     if (xml.name() == "transform")
1311     {
1312         transform = ParseTransform();
1313     }
1314 }
1315 }
1316 xml.readNext();
1317 }
1318 }
1319 Transform t = Transform();
1320 DomeLight* domeLight = new DomeLight(transform, Colour(1), texture,
1321                                         multiplier);
1322 renderer.lights.push_back(domeLight);
1323 }
1324 }
1325
1326 inline void XMLParser::ParseIESLight()
1327 {
1328     xml.readNext();
1329     float multiplier = 1.0f;
1330     Colour albedo(1.0f);
1331     Transform transform;
1332     IESData data;
1333     bool valid = false;
1334     while (!(xml.name() == "light" && xml.isEndElement()))
1335     {
1336         if (xml.isStartElement())
1337         {
1338             qDebug() << "\t" << xml.name();
1339             auto attributes = xml.attributes();
1340             if (xml.name() == "file")
1341             {
1342                 if (attributes.hasAttribute("value"))
1343                 {
1344                     LoadIES(xml.attributes().value("value").toString().
1345                             toStdString(), data);
1346                     valid = true;
1347                 }
1348             }
1349             if (xml.name() == "albedo")
1350             {
1351                 if (attributes.hasAttribute("value"))
1352                 {

```

```

1352         float x, y, z;
1353         ReadFloat3("value", x, y, z);
1354         albedo = Colour(x, y, z);
1355     }
1356 }
1357 if (xml.name() == "multiplier")
1358 {
1359     ReadFloat("value", multiplier);
1360 }
1361 if (xml.name() == "transform")
1362 {
1363     transform = ParseTransform();
1364 }
1365 }
1366 xml.readNext();
1367 }
1368 if (valid)
1369 {
1370     IESLight* ies = new IESLight(transform, albedo, data,
1371         multiplier);
1372     renderer.lights.push_back(ies);
1373 }
1374 }
1375 }
1376 }
1377
1378 inline void XMLParser::ParsePointLight()
1379 {
1380     xml.readNext();
1381     float multiplier = 5000.0f;
1382     Colour albedo(1.0f);
1383     // local to world transform
1384     Transform transform;
1385     while (!(xml.name() == "light" && xml.isEndElement()))
1386     {
1387         if (xml.isStartElement())
1388         {
1389             qDebug() << "\t" << xml.name();
1390             auto attributes = xml.attributes();
1391
1392             if (xml.name() == "albedo")
1393             {
1394                 if (attributes.hasAttribute("value"))
1395                 {
1396                     float x, y, z;
1397                     ReadFloat3("value", x, y, z);
1398                     albedo = Colour(x, y, z);
1399                 }
1400             }
1401             if (xml.name() == "multiplier")
1402             {

```

```

1403             ReadFloat("value", multiplier);
1404         }
1405
1406         if (xml.name() == "transform")
1407     {
1408             transform = ParseTransform();
1409         }
1410     }
1411     xml.readNext();
1412 }
1413 }
1414
1415 PointLight* light = new PointLight(transform, albedo, multiplier);
1416 renderer.lights.push_back(light);
1417 }
1418
1419 inline void XMLParser::ParseSpotLight()
1420 {
1421     xml.readNext();
1422     float tightness = 0.0f;
1423     Colour albedo(5000.0f);
1424     float hotspot = 20.0f;
1425     float falloff = 60.0f;
1426     // local to world transform
1427     Transform transform;
1428     while (!(xml.name() == "light" && xml.isEndElement()))
1429     {
1430         if (xml.isStartElement())
1431     {
1432         qDebug() << "\t" << xml.name();
1433         auto attributes = xml.attributes();
1434
1435         if (xml.name() == "albedo")
1436     {
1437             if (attributes.hasAttribute("value"))
1438         {
1439                 float x, y, z;
1440                 ReadFloat3("value", x, y, z);
1441                 albedo = Colour(x, y, z);
1442             }
1443         }
1444         if (xml.name() == "tightness")
1445     {
1446             ReadFloat("value", tightness);
1447         }
1448         if (xml.name() == "hotspot")
1449     {
1450             ReadFloat("value", hotspot);
1451         }
1452         if (xml.name() == "falloff")
1453     {
1454             ReadFloat("value", falloff);

```

```

1455     }
1456     if (xml.name() == "transform")
1457     {
1458         transform = ParseTransform();
1459     }
1460 }
1461 xml.readNext();
1462 }
1463 }
1464
1465 SpotLight* light = new SpotLight(transform, albedo, hotspot,
1466     falloff, tightness);
1467 renderer.lights.push_back(light);
1468 }
1469
1470 inline void XMLParser::ParseDistantDiscLight()
1471 {
1472     xml.readNext();
1473     Colour albedo(1.0f);
1474     // in degrees
1475     float angle = 20.0f;
1476     // local to world transform
1477     Transform transform;
1478     while (!(xml.name() == "light" && xml.isEndElement()))
1479     {
1480         if (xml.isStartElement())
1481         {
1482             qDebug() << "\t" << xml.name();
1483             auto attributes = xml.attributes();
1484
1485             if (xml.name() == "albedo")
1486             {
1487                 if (attributes.hasAttribute("value"))
1488                 {
1489                     float x, y, z;
1490                     ReadFloat3("value", x, y, z);
1491                     albedo = Colour(x, y, z);
1492                 }
1493                 if (xml.name() == "angle")
1494                 {
1495                     ReadFloat("value", angle);
1496                 }
1497
1498                 if (xml.name() == "transform")
1499                 {
1500                     transform = ParseTransform();
1501                 }
1502             }
1503             xml.readNext();
1504         }

```

```

1506
1507     DistantDiscLight* light = new DistantDiscLight(transform, albedo,
1508             angle);
1509     renderer.lights.push_back(light);
1510 }
1511 inline void XMLParser::ParseDirectionalLight()
1512 {
1513     xml.readNext();
1514     Colour albedo(1.0f);
1515     Vector direction(0,-1, 0);
1516     // local to world transform
1517     Transform transform;
1518     while (!(xml.name() == "light" && xml.isEndElement()))
1519     {
1520         if (xml.isStartElement())
1521         {
1522             qDebug() << "\t" << xml.name();
1523             auto attributes = xml.attributes();
1524
1525             if (xml.name() == "albedo")
1526             {
1527                 if (attributes.hasAttribute("value"))
1528                 {
1529                     float x, y, z;
1530                     ReadFloat3("value", x, y, z);
1531                     albedo = Colour(x, y, z);
1532                 }
1533             }
1534             if (xml.name() == "direction")
1535             {
1536                 float x, y, z;
1537                 ReadFloat3("value", x, y, z);
1538                 direction = Vector(x, y, z);
1539             }
1540
1541             if (xml.name() == "transform")
1542             {
1543                 transform = ParseTransform();
1544             }
1545         }
1546         xml.readNext();
1547     }
1548 }
1549
1550 DistantDirectionalLight* light = new DistantDirectionalLight(
1551     transform, albedo, direction);
1552     renderer.lights.push_back(light);
1553 }
1554 inline void XMLParser::ParseLight()
1555 {

```

```

1556     Q_ASSERT(xml.isStartElement() && xml.name() == "light");
1557     QString type = xml.attributes().value("type").toString();
1558     qDebug() << type << "->";
1559     if (type == "area")
1560     {
1561         ParseAreaLight();
1562     }
1563     if (type == "dome")
1564     {
1565         ParseDomeLight();
1566     }
1567     if (type == "ies")
1568     {
1569         ParseIESLight();
1570     }
1571     if (type == "point")
1572     {
1573         ParsePointLight();
1574     }
1575     if (type == "spot")
1576     {
1577         ParseSpotLight();
1578     }
1579     if (type == "distantdisc")
1580     {
1581         ParseDistantDiscLight();
1582     }
1583     if (type == "directional")
1584     {
1585         ParseDirectionalLight();
1586     }
1587 }
1588
1589 inline void XMLParser::ParseAreaLight()
1590 {
1591     xml.readNext();
1592     Primitive* geometry = nullptr;
1593     Colour c;
1594     while (!(xml.name() == "light" && xml.isEndElement()))
1595     {
1596         if (xml.isStartElement())
1597         {
1598             qDebug() << "\t" << xml.name();
1599             if (xml.name() == "geometry")
1600             {
1601                 geometry = ParseGeometry();
1602             }
1603             if (xml.name() == "colour")
1604             {
1605                 float x, y, z;
1606                 ReadFloat3("value", x, y, z);
1607                 qDebug() << x << y << z;

```

```

1608             c = Colour(x, y, z);
1609         }
1610     }
1611     xml.readNext();
1612 }
1613 }
1614
1615 Transform t = Transform();
1616 DiffuseAreaLight* areaLight = new DiffuseAreaLight(t, c, ((
    GeometricPrimitive*)geometry)->GetGeometry(), geometry->
    GetMaterial());
1617 renderer.lights.push_back(dynamic_cast<Light*>(areaLight));
1618
1619 }
1620
1621 inline void XMLParser::ParseRandomSampler()
1622 {
1623     xml.readNext();
1624     while (!(xml.name() == "sampler" && xml.isEndElement()))
1625     {
1626         if (xml.isStartElement())
1627         {
1628             qDebug() << xml.name() << "->";
1629             if (xml.name() == "sampleCount")
1630             {
1631                 int samples = xml.attributes().value("value").toInt();
1632                 renderer.sampler = new RandomSampler(samples);
1633             }
1634         }
1635         xml.readNext();
1636     }
1637 }
1638
1639 inline void XMLParser::ParseSampler()
1640 {
1641     Q_ASSERT(xml.isStartElement() && xml.name() == "sampler");
1642     QString type = xml.attributes().value("type").toString();
1643
1644     if (type == "random")
1645     {
1646         ParseRandomSampler();
1647     }
1648 }
1649
1650 inline PhaseFunction* XMLParser::ParsePhaseFunction()
1651 {
1652     QString type = xml.attributes().value("type").toString();
1653     if (type == "isotropic")
1654     {
1655         return new Isotropic();
1656     }
1657     if (type == "hg")

```

```

1658     {
1659         float g = 0.0f;
1660         if(xml.attributes().hasAttribute("g"))
1661             g = xml.attributes().value("g").toFloat();
1662         return new HenyeyGreenstein(g);
1663     }
1664
1665     return new Isotropic();
1666 }
1667
1668 inline Volume * XMLParser::ParseHomogeneousVolume()
1669 {
1670     Colour albedo(1.0f);
1671     Colour sigmaA(0.05f);
1672     Colour sigmaS(1.0f);
1673     Colour emission(0.0f);
1674     Point min(0.0f, 0.0f, 0.0f);
1675     Point max(0.0f, 0.0f, 0.0f);
1676     PhaseFunction* phaseFunction = new Isotropic();
1677     Transform* localToWorld = new Transform();
1678     xml.readNext();
1679     while (!(xml.name() == "volume" && xml.isEndElement()))
1680     {
1681         if (xml.isStartElement())
1682         {
1683             auto attributes = xml.attributes();
1684             if (xml.name() == "albedo")
1685             {
1686                 if (attributes.hasAttribute("value"))
1687                 {
1688                     float x, y, z;
1689                     ReadFloat3("value", x, y, z);
1690                     albedo = Colour(x, y, z);
1691                 }
1692             }
1693             if (xml.name() == "sigmaA")
1694             {
1695                 if (attributes.hasAttribute("value"))
1696                 {
1697                     float x, y, z;
1698                     ReadFloat3("value", x, y, z);
1699                     sigmaA = Colour(x, y, z);
1700                 }
1701             }
1702             if (xml.name() == "sigmaS")
1703             {
1704                 if (attributes.hasAttribute("value"))
1705                 {
1706                     float x, y, z;
1707                     ReadFloat3("value", x, y, z);
1708                     sigmaS = Colour(x, y, z);
1709                 }

```

```

1710 }
1711 if (xml.name() == "emission")
1712 {
1713     if (attributes.hasAttribute("value"))
1714     {
1715         float x, y, z;
1716         ReadFloat3("value", x, y, z);
1717         emission = Colour(x, y, z);
1718     }
1719 }
1720 if (xml.name() == "bbox")
1721 {
1722     if (attributes.hasAttribute("min"))
1723     {
1724         float x, y, z;
1725         ReadFloat3("min", x, y, z);
1726         min = Point(x, y, z);
1727     }
1728     if (attributes.hasAttribute("max"))
1729     {
1730         float x, y, z;
1731         ReadFloat3("max", x, y, z);
1732         max = Point(x, y, z);
1733     }
1734 }
1735 if (xml.name() == "phase")
1736 {
1737     qDebug() << "hree";
1738     phaseFunction = ParsePhaseFunction();
1739 }
1740 if (xml.name() == "transform")
1741 {
1742     localToWorld = new Transform(ParseTransform());
1743 }
1744 }
1745 xml.readNext();
1746 }
1747 return new ConstantVolume(localToWorld, albedo, sigmaA, sigmaS,
1748                           emission, BBox(min, max), phaseFunction);
1749 }
1750 inline Volume* XMLParser::ParseGridVolume()
1751 {
1752     Colour albedo(1.0f);
1753     Colour sigmaA(0.05f);
1754     Colour sigmaS(1.0f);
1755     Colour emission(0.0f);
1756     Colour temperature(0.0f);
1757     PhaseFunction* phaseFunction = new Isotropic();
1758     Transform* localToWorld = new Transform();
1759     float densityMultiplier = 1.0f;
1760     xml.readNext();

```

```

1761 VolumeData data;
1762
1763 while (!xml.name() == "volume" && xml.isEndElement())
1764 {
1765     if (xml.isStartElement())
1766     {
1767         auto attributes = xml.attributes();
1768         if (xml.name() == "file")
1769         {
1770             if (attributes.hasAttribute("value"))
1771             {
1772                 LoadVolumeData(data, const_cast<char*>(xml.
1773                         attributes().value("value").toString().
1774                         toStdString().c_str()));
1775             }
1776             if (xml.name() == "albedo")
1777             {
1778                 if (attributes.hasAttribute("value"))
1779                 {
1780                     float x, y, z;
1781                     ReadFloat3("value", x, y, z);
1782                     albedo = Colour(x, y, z);
1783                 }
1784                 if (xml.name() == "sigmaA")
1785                 {
1786                     if (attributes.hasAttribute("value"))
1787                     {
1788                         float x, y, z;
1789                         ReadFloat3("value", x, y, z);
1790                         sigmaA = Colour(x, y, z);
1791                     }
1792                 }
1793                 if (xml.name() == "sigmaS")
1794                 {
1795                     if (attributes.hasAttribute("value"))
1796                     {
1797                         float x, y, z;
1798                         ReadFloat3("value", x, y, z);
1799                         sigmaS = Colour(x, y, z);
1800                     }
1801                 }
1802                 if (xml.name() == "emission")
1803                 {
1804                     if (attributes.hasAttribute("value"))
1805                     {
1806                         float x, y, z;
1807                         ReadFloat3("value", x, y, z);
1808                         emission = Colour(x, y, z);
1809                     }
1810                 }

```

```

1811     if (xml.name() == "temperature")
1812     {
1813         if (attributes.hasAttribute("value"))
1814         {
1815             float x, y, z;
1816             ReadFloat3("value", x, y, z);
1817             temperature = Colour(x, y, z);
1818         }
1819     }
1820     if (xml.name() == "multiplier")
1821     {
1822         if (attributes.hasAttribute("value"))
1823         {
1824             ReadFloat("value", densityMultiplier);
1825         }
1826     }
1827     if (xml.name() == "phase")
1828     {
1829         phaseFunction = ParsePhaseFunction();
1830     }
1831     if (xml.name() == "transform")
1832     {
1833         localToWorld = new Transform(ParseTransform());
1834     }
1835 }
1836     xml.readNext();
1837 }
1838     return new VolumeGrid(localToWorld, albedo, sigmaA, sigmaS,
1839                           emission, temperature, data, phaseFunction, densityMultiplier);
1840 }
1841 inline void XMLParser::ParseVolumeIntegrator()
1842 {
1843     QString type = xml.attributes().value("type").toString();
1844     qDebug() << type << "->";
1845     xml.readNext();
1846     float stepSize = 1.0f;
1847     VolumeIntegrator* volumeIntegrator = new RayMarcher(stepSize);
1848     if (xml.attributes().hasAttribute("stepSize"))
1849         ReadFloat("stepSize", stepSize);
1850
1851     if (type == "homogeneous")
1852         renderer.volume = ParseHomogeneousVolume();
1853     if (type == "grid")
1854         renderer.volume = ParseGridVolume();
1855     renderer.volumeIntegrator = volumeIntegrator;
1856 }
1857
1858 inline Renderer* XMLParser::Read(QIODevice* device)
1859 {
1860     xml.setDevice(device);

```

```

1862     while (!xml.atEnd()) {
1863         QXmlStreamReader::TokenType token = xml.readNext();
1864
1865         if (token == QXmlStreamReader::StartElement)
1866         {
1867             if (xml.name() == "camera")
1868                 renderer.camera = ParseCamera();
1869             else if (xml.name() == "geometry")
1870                 ParseGeometry();
1871             else if (xml.name() == "common")
1872                 ParseCommonSettings();
1873             else if (xml.name() == "integrator")
1874                 ParseIntegrator();
1875             else if (xml.name() == "light")
1876                 ParseLight();
1877             else if (xml.name() == "sampler")
1878                 ParseSampler();
1879             else if (xml.name() == "volume")
1880                 ParseVolumeIntegrator();
1881         }
1882
1883         if (xml.hasError())
1884             qDebug() << "xml parse error:" << xml.errorString() << xml.
1885             lineNumber() << xml.columnNumber();
1886         return nullptr;
1887     }
1888
1889     renderer.bitmap = new Bitmap(renderer.width, renderer.height);
1890 //renderer.integrator = new WhittedIntegrator(25);
1891     renderer.scene = new Scene(renderer.primitives, renderer.lights,
1892         renderer.integrator, new BVH(renderer.primitives), renderer.
1893         camera);
1894     renderer.scene->SSpecularGeometries = renderer.specularSurfaces;
1895     if (renderer.volumeIntegrator != nullptr)
1896     {
1897         renderer.scene->SVolumeIntegrator = renderer.volumeIntegrator;
1898         renderer.scene->SVolumes = renderer.volume;
1899         //VolumeData data;
1900         //LoadVolumeData(data, R"(C:\Users\k1332225\Documents\GitHub\
1901             photon-mapping\PhotonMapping\Scenes\test.volume)");
1902         //VolumeGrid* g = new VolumeGrid(new Transform(), Colour(1),
1903             Colour(0.05), Colour(.5), Colour(0), data);
1904         //renderer.scene->SVolumes = g;
1905     }
1906     if (!renderer.IsValid())
1907     {
1908         return nullptr;
1909     }
1910     renderer.integrator->Preprocess(renderer.scene, renderer.sampler);
1911     return new TileRenderer(renderer.scene, renderer.camera, renderer.
1912         integrator, renderer.bitmap, renderer.sampler, 32);

```

