Lyrae FX Ray-Tracing Renderer

Technical Documentation

赵若旭	5070309078	F0703020	zhaoruoxu@gmail.com
陈美先	5070309339	F0703010	misamchen@sjtu.edu.cn
张钰琦	5070309730	F0703027	kid1125@hotmail.com
张晴	5070309457	F0703015	sjtuzhq@gmail.com

Table of Content

1. Introduction 3
   - About Lyrae FX 3
   - Design Goals 3
   - Feature List 3
2. System Overview 5
   - Architecture 5
   - Flow of Execution 6
3. Geometry and Shapes 10
   - Vector and Transform 10
   - Coordinate System 10
   - Ray-Shape Intersection 10
   - Sphere 11
   - Box 11
   - TCylinder 12
   - Cylinder 13
   - Triangle and TriangleMesh 14
   - Accelerators 15
     - Grid Accelerator 15
     - Bounding Volume Hierarchy 15
4. Color and Materials 17
   - Color Representation 17
   - Materials and BSDF 17
     - BSDF Types and Interface 17
     - Lambertian 18
     - Phong 19
     - Fresnel Reflectance 20
     - Specular Reflection 21
     - Specular Transmission (Refraction) 21
     - Glossy Reflection 22
     - Glossy Transmission 23
     - Real World Materials 24
5. Camera Models 26
   - Orthographic camera 26
   - Perspective camera 26
6. Sampling 28
   - Supersampling 28
   - Adaptive supersampling 29
   - Importance Sampling 30
7. Lights 31
   - Point Light 31
   - Area Light 32
   - Spot Light 33

Shadows....33
8. Texture....34
Texture Mapping....34
Sphere....34
Box....34
TriangleMesh....35
Texture Filtering....36
9. Monte Carlo Methods and Distributed Ray Tracing....37
Soft Shadow....37
Depth of Field....38
Glossy Reflection and Transmission....40
10. Global Illumination....42
Instant Global Illumination (IGI)....42
Ambient Occlusion....43
Photon Mapping....45
11. Volumetric Lighting....50
12. Post Processing....52
13. Parallelism....53
14. Performance....54
15. References....58
Appendix A: 3rd Party Libraries....59
Appendix B: Scene Description File Format....60

1. Introduction

About Lyrae FX

Lyrae FX is a rendering engine based on ray tracing algorithms, targeting at generating high quality images. It is capable to simulate the physical behavior of light interacting with the environment. It uses distributed ray tracing algorithms to produce effect such as soft shadow and depth of field. It can also simulate global illumination effects such as indirect diffuse illumination and caustics.
Here's some pictures rendered by Lyrae FX.

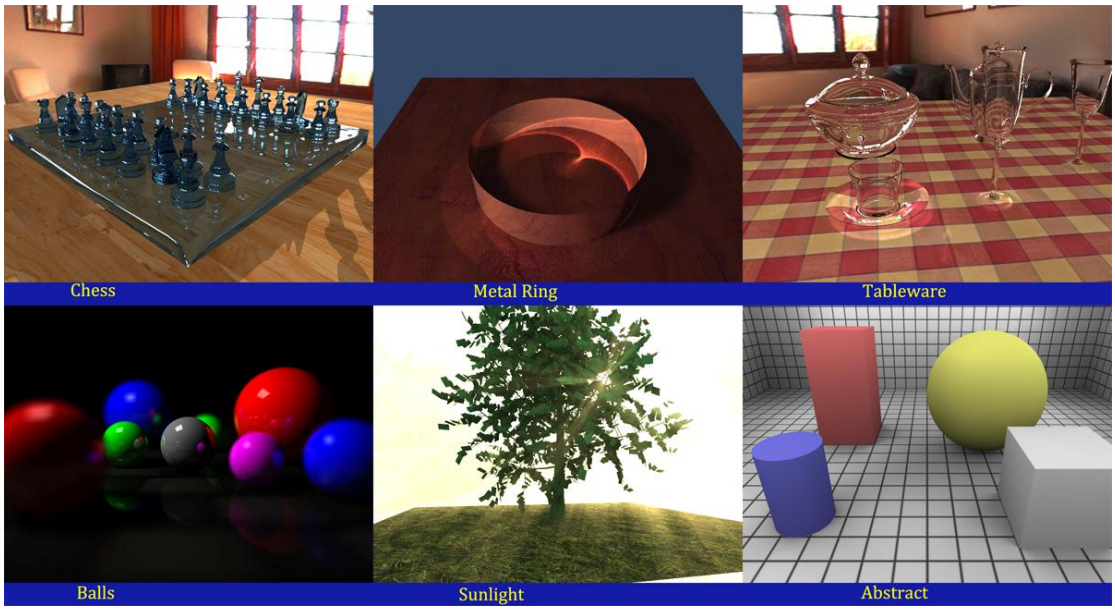

Figure 1 Images rendered by Lyrae FX

Design Goals

Lyrae FX is designed to achieve the following goals:

- Capability to produce photo-realistic images
- Extendible programming framework, easy to add new features
- High rendering efficiency to ensure acceptable performance
- Flexible configuration and description files
- Friendly user interface

Feature List

Here's a list of current features supported by Lyrae FX:

- Full support of basic built-in shapes and triangle meshes
- Materials described by BSDF
- Glossy reflection and transmission
- Adaptive sampling
- Perspective/orthographic camera models
- Texture mapping
- Bilinear texture filtering
- Soft shadow
- Depth of field
- Instant global illumination
- Ambient Occlusion
- Indirect diffuse illumination by photon mapping
- Caustics by photon mapping
- Volumetric Lighting
- Tone mapping and HDR pipeline
- Parallel rendering
- 3DS file loading
- Simple scene description file using XML

Here's a GUI screenshot of Lyrae FX:

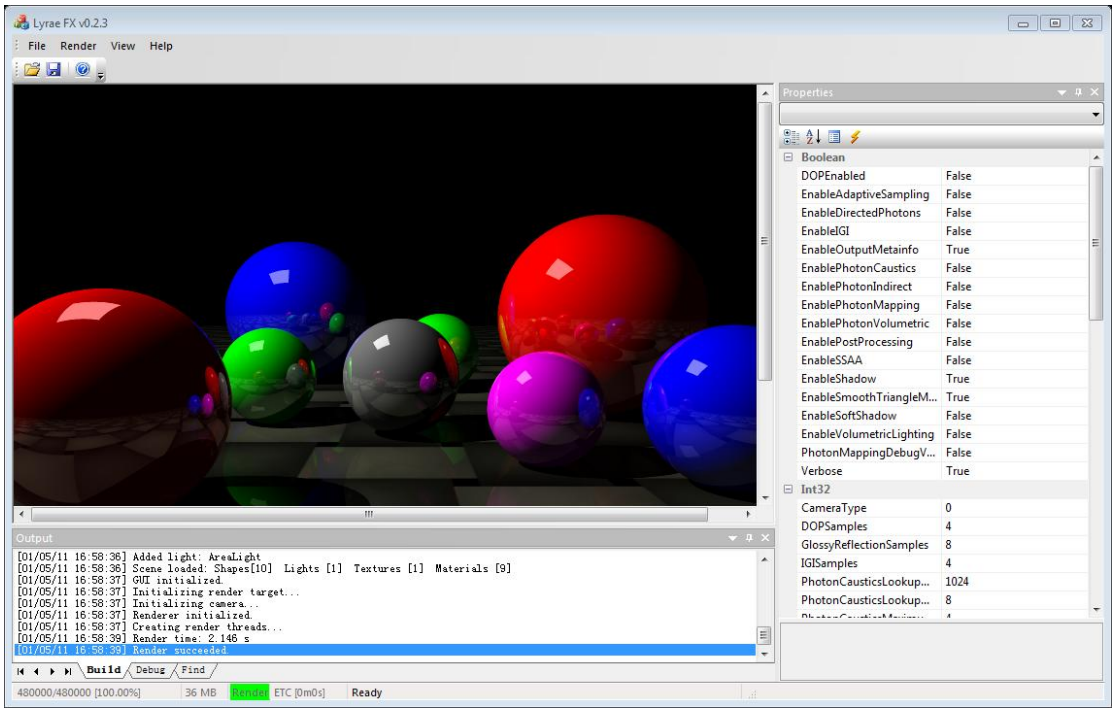

Figure 2 GUI Screenshot of Lyrae FX

2. System Overview

Architecture

Lyrae FX is implemented using object-oriented approach. The rendering framework is constructed by the following main classes:

- **Engine**: Defines the main rendering loop; also contains other classes that forms the renderer,
- **Scene**: Contains the current rendering scene, including shapes, materials, lights, textures, accelerators, etc.;
- **Material**: Describes a kind of material; contains one or more BSDF classes to represent the light scattering behavior described by Bidirectional Scattering Distribution Function;
- **Shape**: Contains the geometry information about a single primitive in the scene, such as a sphere or a triangle mesh; it also contains a pointer to Material class to represent its material;
- **Light**: Provides interface to describe a kind of light;
- **Texture**: Loads, stores and samples from a texture;
- **PostProcessor**: Provides interface to implement a post processor; A post processor is used to do post processing, such as tone mapping;
- **BSDF**: Represents a sub-material described by Bidirectional Scattering Distribution Function; provides interface of E(), F() and G();(explained later)
- **PhotonMap**: Stores photon maps and provides methods to do photon mapping
- **BVHAccelerator**: Accelerator using Bounding Volume Hierarchy;
- **RenderTarget**: Stores the rendered image; provides conversions between HDR and RGB image formats;
- **Task**: Stores information of a task for parallel running;
- **Random**: a pseudorandom number generator;
- **Config**: Stores configuration information

- **Vector**, **Matrix**, **Color**, **Ray**, **AABB**: basic geometry and shading classes;
- **TriangleMesh**, **Box**, **Sphere**, **Cylinder**, **TCylinder**, **Plane**: subclasses of **Shape**;
- **PointLight**, **AreaLight**, **SpotLight**, **VirtualLight**: subclasses of **Light**;
- **Lambertian**, **Phong**, **SpecularReflection**, **SpecularTransmission**, **SimpleLight**: subclasses of **BSDF**;
- **ToneMapping**, **BlurPostProcessor**, **HDRPostProcessor**: subclasses of **PostProcessor**;

Other miscellaneous classes are omitted, they are not as important as the classes mentioned above. The relations of each class are described in the following graph.

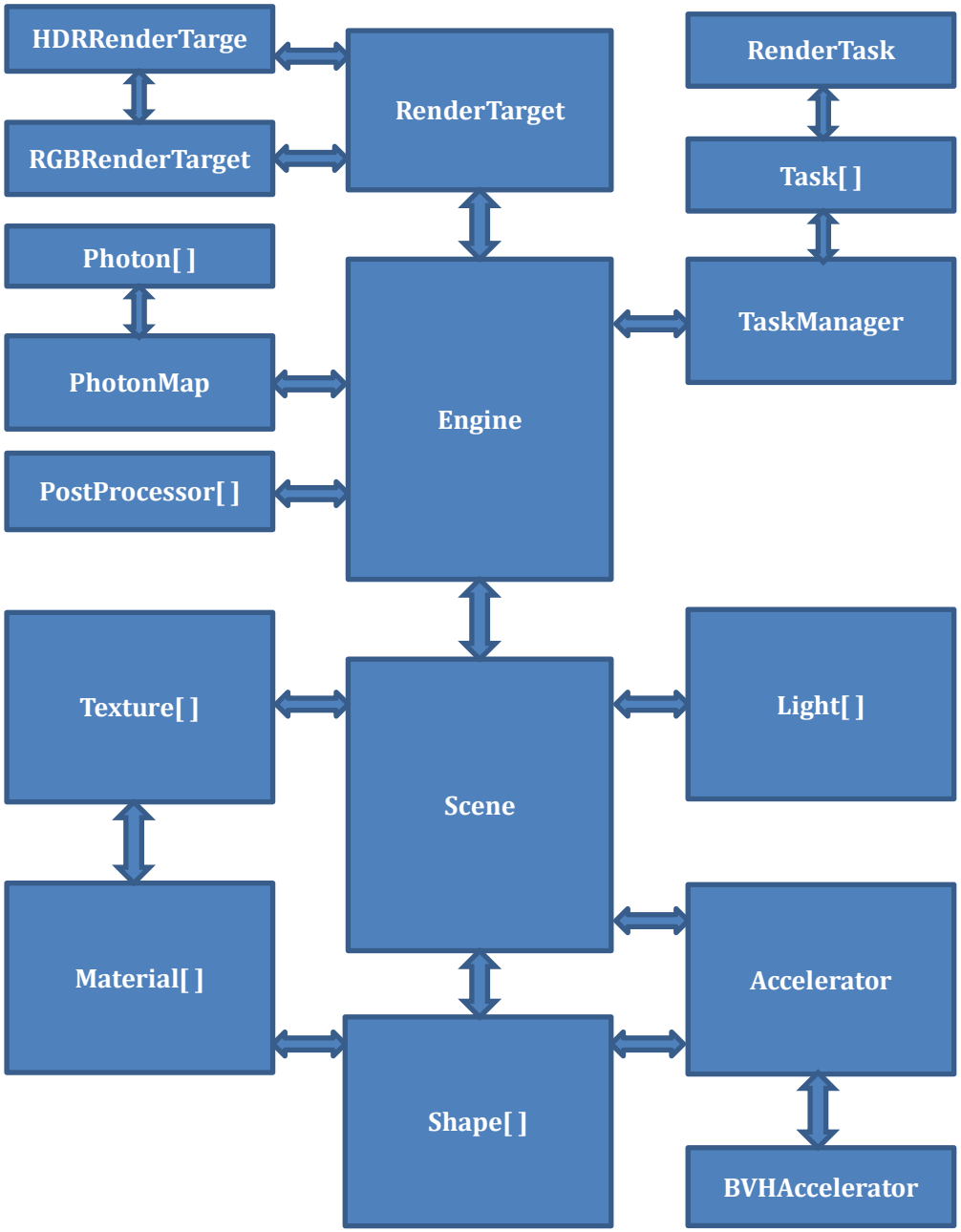

Figure 3 Lyrae FX Architecture

Flow of Execution

The rendering flow of Lyrae FX has four main phases: 0) scene building; 1) pre-processing; 2) per-pixel rendering; 3) post-processing. Each phase has the following purposes:

0) **Scene Building**: Reads the XML scene description file, loads textures and 3DS files; load renderer configurations; builds the (BVH) accelerator;
1) **Pre-processing**: creates rendering tasks; builds photon maps for global illumination and caustics, or builds instant global illumination virtual lights;
2) **Per-pixel rendering**: Shoot rays for every pixel, performs ray-object intersection calculating, evaluates the material's BSDF and illumination; recursively trace rays for

reflection and transmission;

3) **Post-processing**: Synthesizes image from the raw traced illumination information, displays the image on screen and saves to disk, cleans up garbage produced by the renderer.

The quality of final image is mainly decided by the second and third phase.

The execution flow of each phase is explained in the following diagrams:

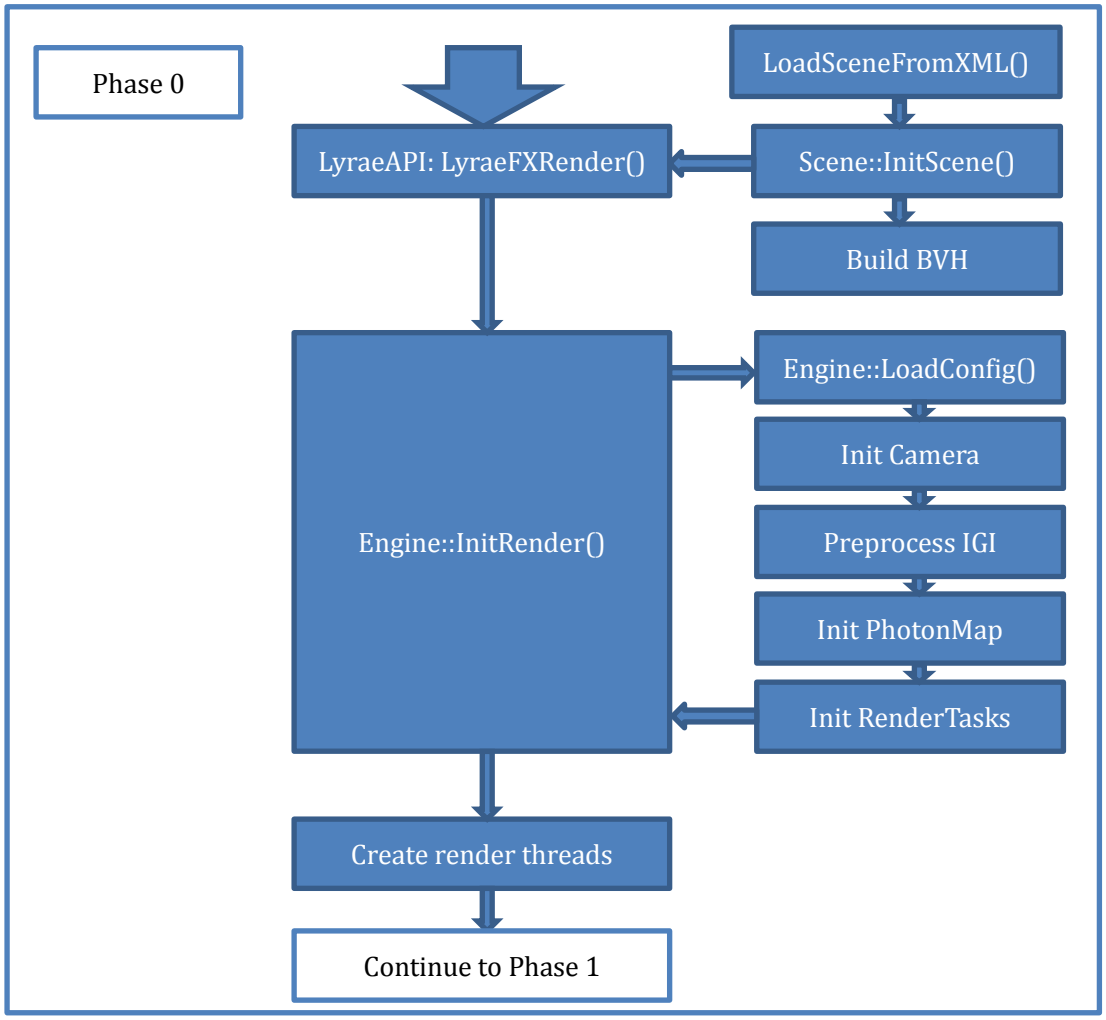

Figure 4 Lyrae FX Render Phase 0

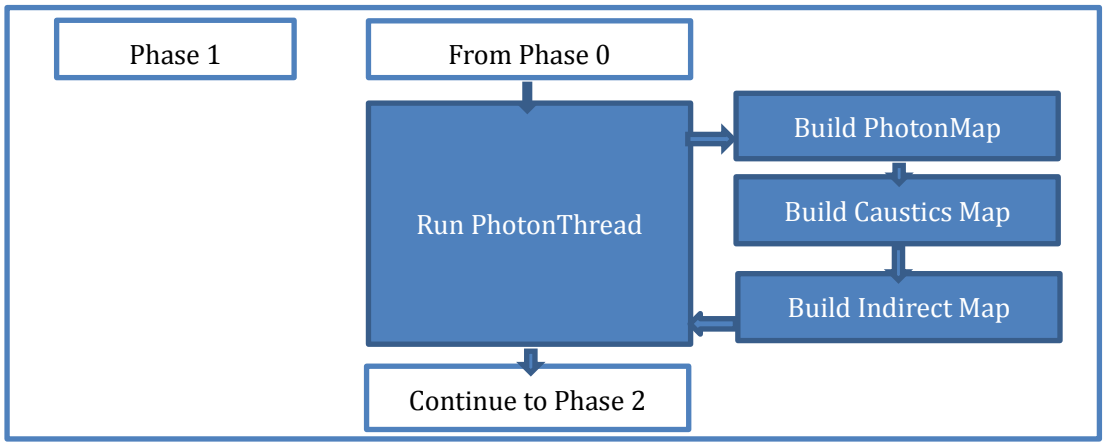

Figure 5 Lyrae FX Render Phase 1

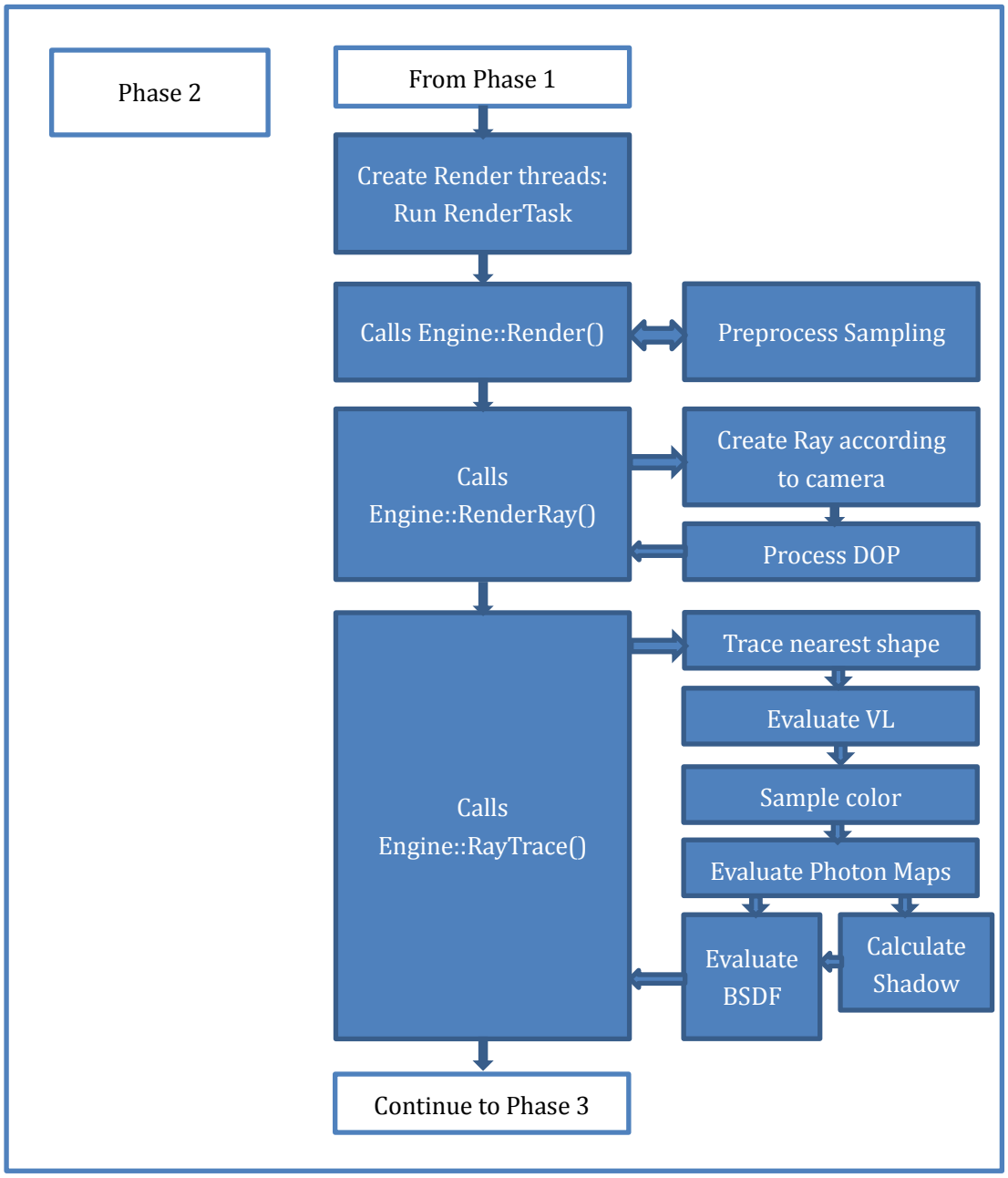

Figure 6 Lyrae FX Render Phase 2

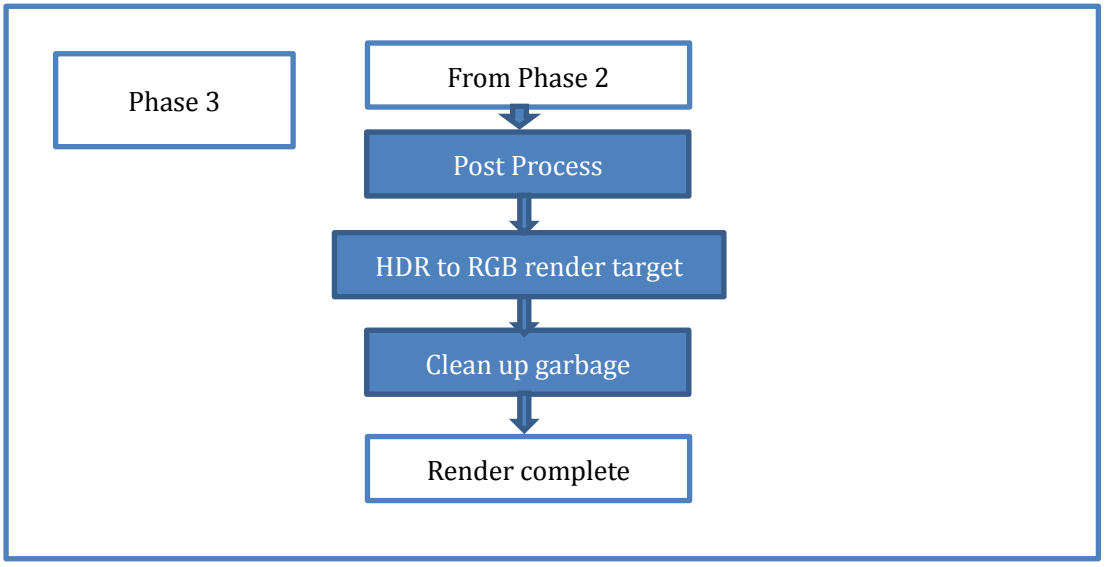

Figure 7 Lyrae FX Render Phase 3

Image saving and displaying is handled by GUI part of Lyrae FX. These implementation details are omitted here.

3. Geometry and Shapes

The geometry part is the fundamental representation of mathematical concepts, including vector calculation, matrix calculation and transform, primitive equations, ray-shape intersection, etc.
For simplicity, Lyrae FX does not distinguish vector, point and normal. They are all represented by a single class Vector. The fundamental classes for geometry are **Vector**, **Matrix**, **Ray**, **Pln**(plane abstraction), **AABB**(Axis-Aligned Bounding Box).

Vector and Transform

Vectors are defined by class **Vector**, which contains three float values(x, y, z, 12 bytes). The Vector class also defines basic mathematical operations between Vectors, such as dot product.
Matrices are defined by class **Matrix**, which has a degree of 4x4, thus contains 16 float values (64 bytes). Matrices are used to transform Vectors.
See Core/Geometry.h(.cpp) for implementation details.

Coordinate System

Lyrae FX uses a right-handed coordinate system, as shown below:

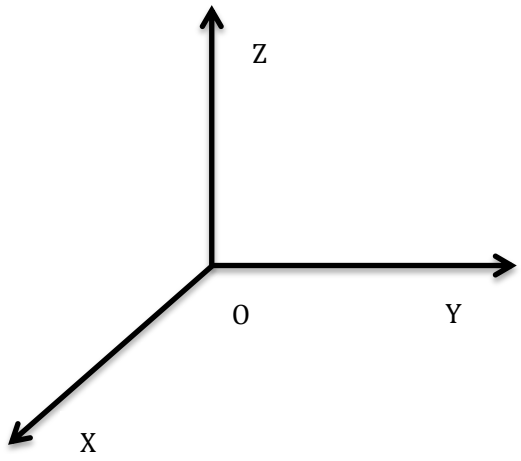

Figure 8 Lyrae FX Coordinate System

Ray-Shape Intersection

Rays are represented by the Ray class, which contains its origin and direction. Ray-shape intersection is calculated by each shape, with the Shape::Intersect() member function. There're three possible intersecting results:

IR_MISS: ray and shape fail to intersect, or the intersecting distance is larger than the ray's maximum distance;
IR_HIT: ray hits the shape from the outside of shape;
IR_HITINSIDE: ray hits the shape from the inside of shape.
The intersecting result is represented by the distance from the ray's origin to the intersection point.

Sphere

Spheres are represented by the following equation:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2$$

Spheres have two attributes: Position and Radius, and their meanings are obvious.
Ray-Sphere intersection is calculated by substituting Ray(x, y, z) = O + tD to the equation above, and the implementation is in Sphere::Intersect() method.
The normal of Sphere is pointing directly from the centroid to the intersection point.

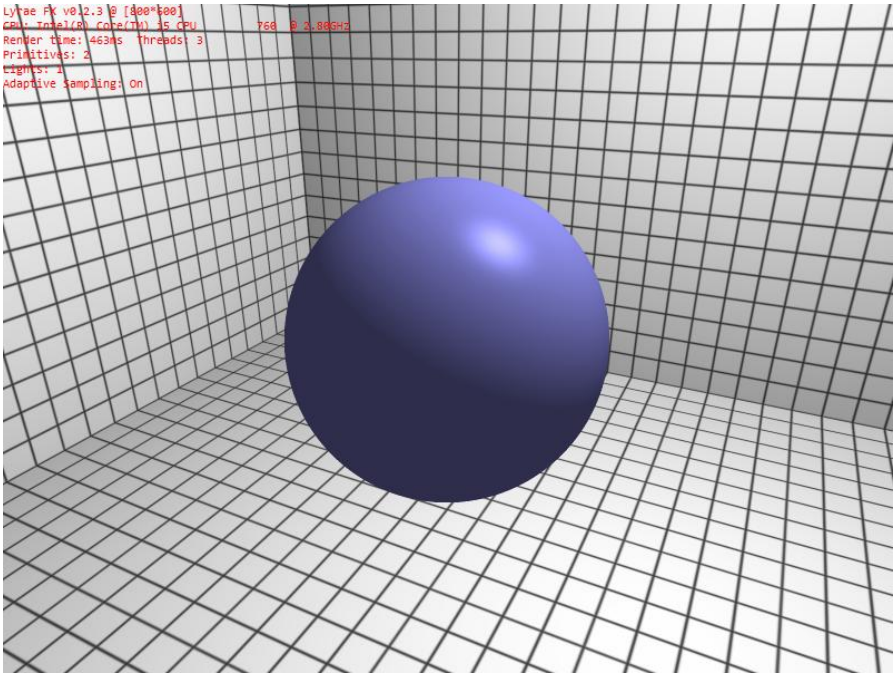

Figure 9 Sphere

Box

Box is the instantiation of the class AABB, which has two attributes: Position and Size. Currently Lyrae FX only supports axis-aligned boxes for simplicity.
Box-Ray intersection is implemented in Box::Intersect() member function.

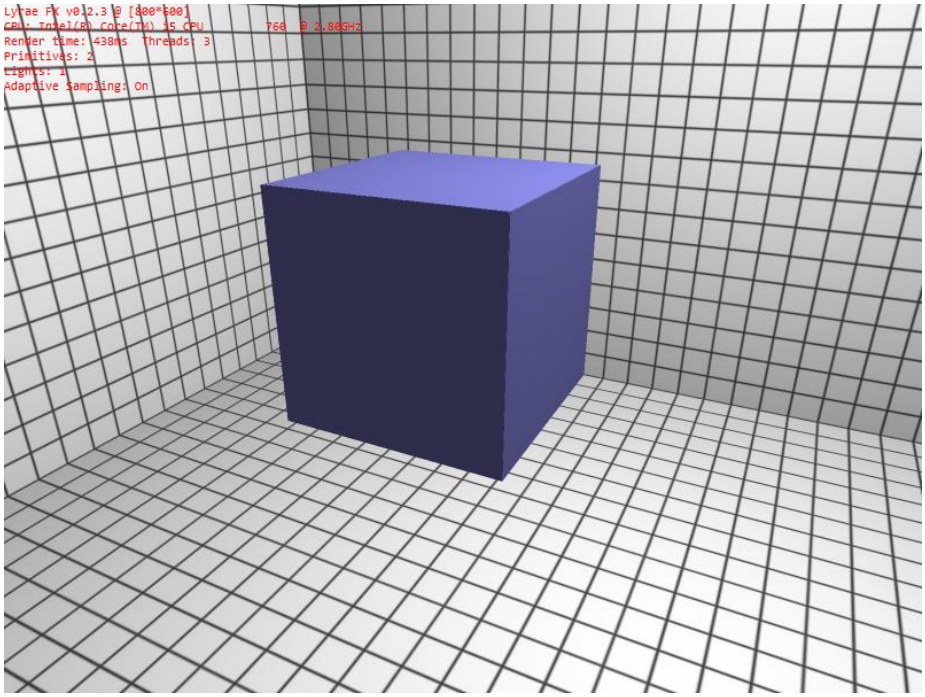

Figure 10 Box

TCylinder

TCylinder represents the 'topless' cylinders, which do not have the top and bottom cover. TCylinders are described by the following equation:

$$(x - x_0)^2 + (y - y_0)^2 = R^2$$

$$|z - z_0| \leq H$$

TCylinders have three attribute: Position, Radius and HalfHeight, representing the R and H in the equation above.

Ray-TCylinder intersection calculation is similar as spheres.

Currently Lyrae FX only supports TCylinders that the central axes aligns to the Z-axis.

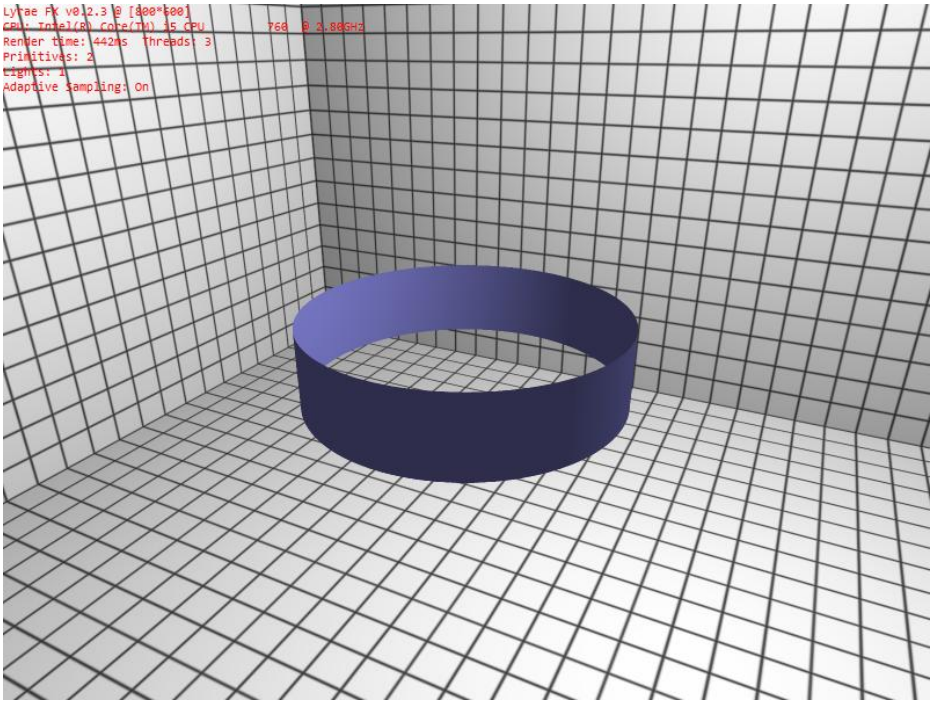

Figure 11 TCylinder

Cylinder

The Cylinder class represents a combination of a TCylinder and its top and bottom covers, which makes the shape solid. Its ray-cylinder calculation is a little complicated than TCylinder, because it has to judge intersections with top and bottom covers.
Cylinders also have two attributes: Position, Radius and HalfHeight.

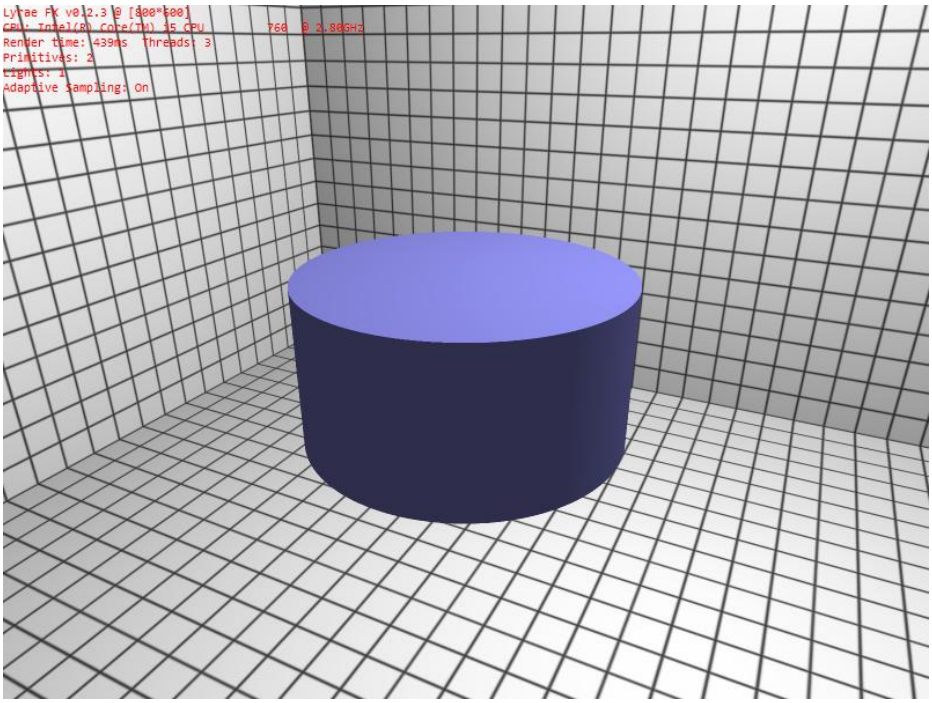

Figure 12 Cylinder

Triangle and TriangleMesh

Triangles are the most basic primitive in most graphics systems. Lyrae FX provides full support for triangles and triangle mesh. A triangle mesh contains from tens to millions of triangles, to construct very complicated models. All other shapes, such as cones, tubes and pyramids, can be represented by triangle meshes.

A triangle has three vertices. Ray-triangle intersection is calculated using the **Barycentric Coordinates**. In this method, each point P in triangle can be represented as:

$$P = a_1 P_1 + a_1 P_2 + a_3 P_3$$

P, P_1, P_2, P_3 are all vectors. By solving the linear equations, we can easily judge if a point is in a triangle by judging if $0 \leq a_1, a_2, a_3 \leq 1$.

The normal of triangle is the cross product of two edges of a triangle, usually P_1P_2 cross P_1P_3. However, in triangle meshes, normal are pre-calculated to get a smooth look of the mesh.

A triangle mesh contains the following data:

int ntris: number of triangles;

int nverts: number of vertices;

int vertexIndex[]: an array of indices of the vertices of each triangle;

Vector p[]: vertices;

Vector n[]: normal;

float uvs[]: triangles' UV coordinates (for texture mapping).

Basically in a triangle mesh, all vertices are stored in an array of Vector. All triangles share this array. A triangle's vertices are specified by the vertices' index. Each vertex has its own normal, which is calculated by averaging all normals of the triangles that share this vertex. a triangles stores a pointer to the triangle mesh, and its starting vertex pointer. The three vertices of a triangle are p[v[0]], p[v[1]], p[v[2]], where **p** is the array of vertices, and **v** is the array of vertex indices.

Lyrae FX support loading triangle mesh from external files with 3DS file format. This is implemented using **lib3ds** 3rd party library.

Here's an image of the famous teapot triangle mesh.

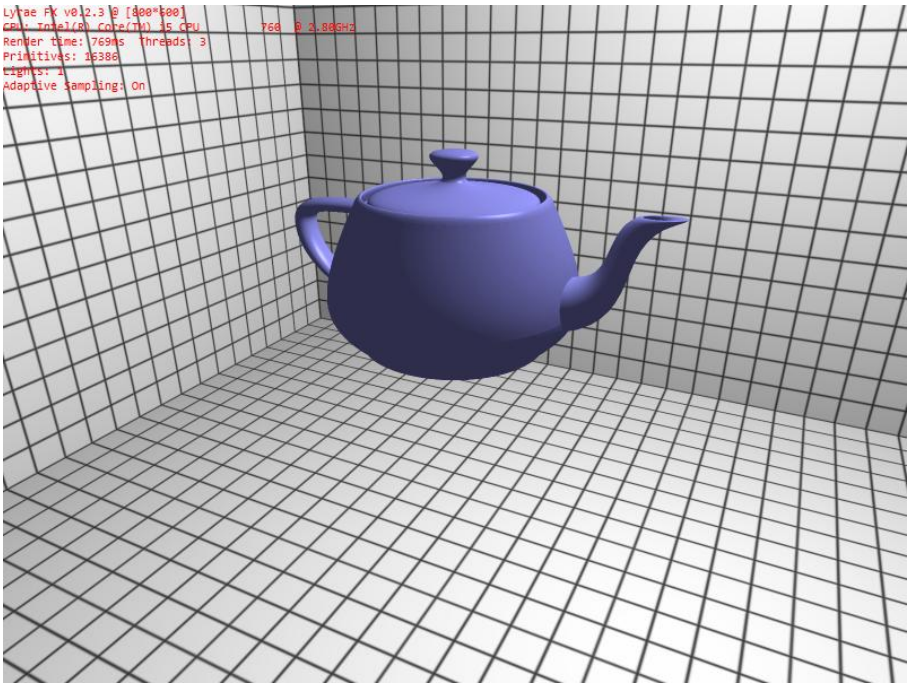

Figure 13 Triangle Mesh consisting of 16,384 triangles

Accelerators

The naïve method of find the nearest shape hit by a ray is to check for intersection of the ray and each shape one by one. Obviously, this method has very low efficiency, because most of the ray-shape intersection calculations are useless.

To speed up the ray intersection calculation by eliminating unnecessary ray-shape intersection calculation, we need to use spatial subdivision algorithm. Generally, we use the class **Accelerator** to perform spatial subdivision. Here we introduce the Grid accelerator, and Bounding Volume Hierarchy algorithm.

Grid Accelerator

Earlier versions of Lyrae FX supports Grid accelerator. The Grid accelerator divide the space into small axis-aligned boxes, each box contains only the shapes that intersect the box. When a ray is traced, it moves from box to box along the ray direction. When a new box is met, only the shapes in the box are checked for intersection.

This algorithm is quite simple, and its efficiency is still not good, when shapes are distributed non-balanced. Current version of Lyrae FX does not support Grid accelerator any more.

Bounding Volume Hierarchy

A bounding volume hierarchy is a tree structure on a set of shapes. All shapes are wrapped in bounding volumes (in Lyrae FX, AABBs) that form the leaf nodes of the tree. These nodes are

then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the root.

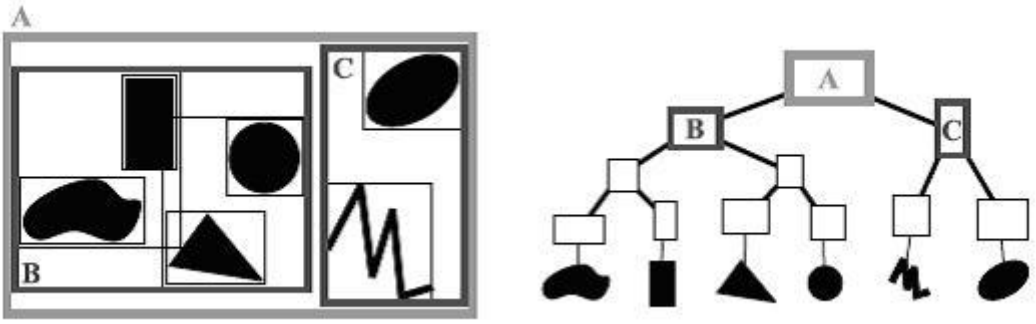

Figure 14 Bounding Volume Hierarchy

Lyrae FX uses Surface Area Heuristic(SAH) for BVH partitioning. The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray-shape intersection tests for a particular partitioning of primitives. Building acceleration structures can follow the goal of minimizing total cost. Using the ideas of geometric probability, the probability of a ray intersecting a shape is related to the shape's surface area. Based on this idea, the surface areas of two volumes are evaluated when partitioning, which gives a good structure of BVH for ray tracing.

For ray-shape intersection tests, the bounding volumes are tested from the root recursively. If the root doesn't intersect with the ray, its two children definitely don't intersect with the ray either. If it does, both its two children are tested for intersection. This process is performed recursively, until an intersection is found or all possible bounding volumes are tested.

The BVH accelerator is implemented in Core/BVH.cpp. The implementation details are omitted here.

4. Color and Materials

Color Representation

Lyrae FX uses a simple approach to represent color: three float values representing the color's R, G, B channels. This color representation is quite simple, but it's enough for light-weighted ray tracers like Lyrae FX.
To eliminate code, the Color is just a typedef of Vector class, since most of their operations are the same.

Materials and BSDF

Lyrae FX describes materials using **Bidirectional Scattering Distribution Function (BSDF)**. The BSDF is used to name the general mathematical function which describes the way in which light is scattered by a surface. In practice this phenomenon is usually split into reflected and transmitted components, which are then treated separately as Bidirectional Reflectance Distribution Function (BRDF) and Bidirectional Transmittance Distribution Function (BTDF). In Lyrae FX, BRDF and BTDF share the same interface provided by class BSDF.
Each material can contain several BSDFs to represent the mixed behavior of light scattering.

BSDF Types and Interface

There are five BSDF types in Lyrae FX, they are:
a) **Reflection**: Describes materials that reflect light;
b) **Transmission**: Describes materials that transmit (refract) light;
c) **Diffuse**: Describes materials that has diffuse reflection or transmission (all directions), or similar behaviors;
d) **Specular**: Describes materials that have specular reflection or transmission (only one direction);
e) **Glossy**: Describes materials that have glossy reflection or transmission (limited directions).
One BSDF must have one of **Reflection** and **Transmission**, and one of **Diffuse**, **Specular** and **Glossy**.
Materials usually contain more than one BSDF to accurately describe the light scattering behaviors. For example, the glass material contains a BSDF with REFLECTION | SPECULAR and a BSDF with TRANSMISSION | SPECULAR, representing both light reflectance and transmittance.
The BSDF class has three main member functions, called **E()**, **F()** and **G()**. Their meanings and usages are described below:

- **E(Vector Vi, Vector N)**

The E function evaluates BSDF's emissive light, such as the simple ambient light and self-illuminant light. It takes parameter Vi as point-to-observer direction and shape's normal N at the sampling point.

- **F(Vector Vi, Vector Vo, Vector N)**
  The F function evaluates BSDF's diffuse reflection. Given the point-to-observer vector Vi, point-to-light vector Vo and shape's normal N at this point, this function calculates the light scattered into the given direction.
- **G(Vector Vi, Vector [] Vo, Vector N)**
  The G function evaluates BSDF's specular and glossy reflection and transmission. Given the light incoming direction Vi (unlike in E and F Vi is the point-to-observer direction), this function calculates the possible outgoing directions, and the light scattered. For specular reflection and transmission, there's one possible outgoing direction. However, for glossy reflection and transmission, there's usually more than one outgoing directions. In this situation, the G function samples a random set with limited number of directions. This is called Monte Carlo method, which will be introduced later in distributed ray tracing section.

With the basic interface introduced, we can describe some of the concrete BSDFs.

Lambertian

Lambertian reflectance model is the simplest reflectance model, given by the following equation:

$$I_D = \boldsymbol{L} \cdot \boldsymbol{N} C I_L,$$

where N is the surface's normal vector, L is the normalized light-direction vector, C is the color and I_L is the intensity of incoming light. I_D is the intensity of the diffusely reflected light. Thus, the E and G functions simply return 0, and F function returns I_D.
Here's a picture demonstrating Lambertian model.

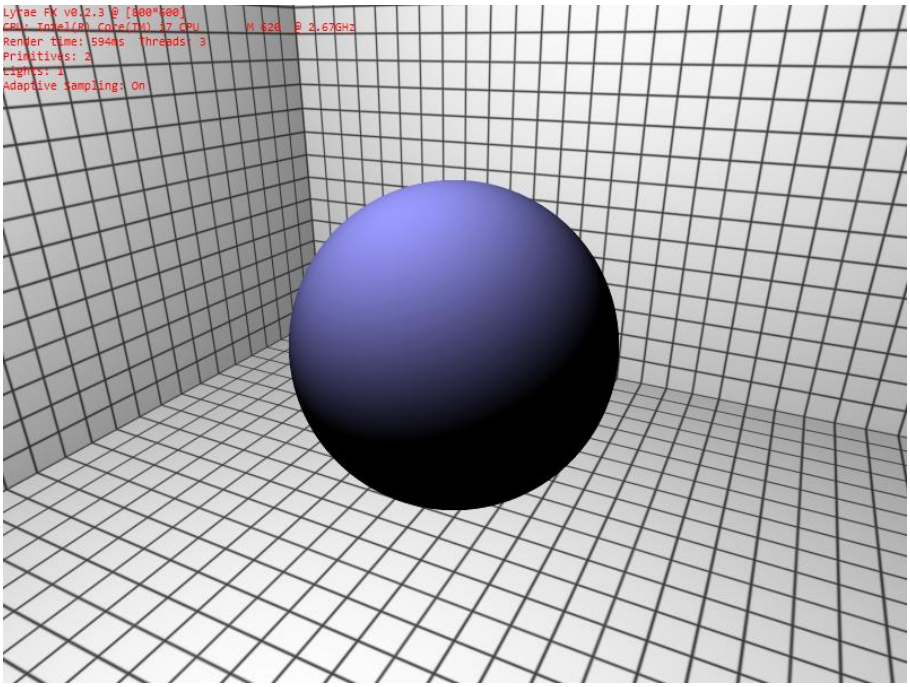

Figure 15 Lambertian model

Phong

Phong reflection model is an empirical model of local illumination. It is a little more complicated than Lambertian model, adding specular highlight and ambient light.
Phong reflection model can be represented by the following equation:

$$\boldsymbol{I_p = k_a i_a + \sum_{m \in lights} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^{\alpha} i_s)}$$

where
$\boldsymbol{k_s}$: specular reflection constant;
$\boldsymbol{k_d}$: diffuse reflection constant;
$\boldsymbol{k_a}$: ambient reflection constant
$\boldsymbol{\alpha}$: shininess constant
$\boldsymbol{L_m}$: direction vector from the point on the surface toward each light source
$\boldsymbol{N}$: normal at this point on the surface
$\boldsymbol{R}$: direction of perfectly reflected ray
$\boldsymbol{V}$: direction pointing towards the observer
The ambient term is evaluated in the E function, and diffuse and specular terms are evaluated in the F function. The G function of Phong model returns zero.
Here's a image rendered using Phong model:

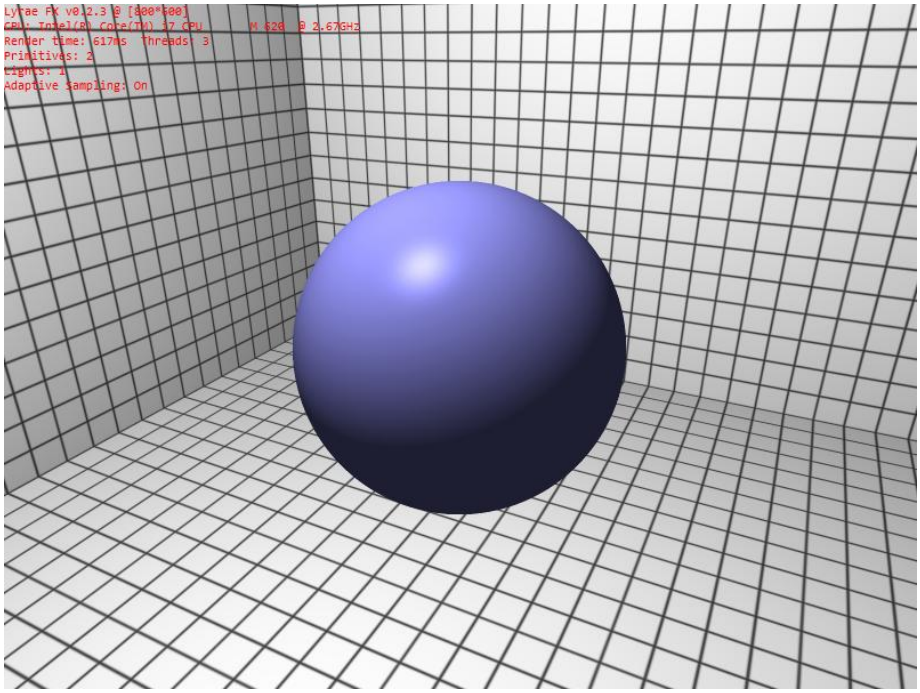

Figure 16 Phong model

Fresnel Reflectance

Before we introduce specular reflection and specular transmission, let's first look at the Fresnel equations. The Fresnel equations describe the behavior of light when moving between media of differing refractive indices. For simple ray tracers, the fraction of incoming light that is reflected or transmitted may be constants. In fact, however, these values are directionally dependent, given by Fresnel equations. A close approximation to Fresnel reflectance formula for dielectrics is:

$$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}$$

$$r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}$$

Where η_i and η_t are the indices of refraction for incident and transmitted media.
The Fresnel reflectance for unpolarized light is

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2)$$

The Fresnel formula for conductors is a little bit different, for conductors don't transmit light. A widely used approximation to the Fresnel reflectance for conductors is

$$r_{\parallel}^2 = \frac{(\eta^2 + k^2)\cos\theta_i{}^2 - 2\eta\cos\theta_i + 1}{(\eta^2 + k^2)\cos\theta_i{}^2 + 2\eta\cos\theta_i + 1}$$

$$r_{\perp}^2 = \frac{(\eta^2 + k^2) - 2\eta\cos\theta_i + \cos\theta_i{}^2}{(\eta^2 + k^2) + 2\eta\cos\theta_i + \cos\theta_i{}^2}$$

These two kinds of Fresnel reflectance are defined in class **FresnelDielectric** and **FresnelConductor**. There's also a special kind of Fresnel type which reflect 100% of the incoming light. This kind of Fresnel reflectance is defined in **FresnelSpecial**.

Specular Reflection

Using the Fresnel reflectance equations, we can now implement the specular reflection model. In specular reflection, light reflect perfectly at the intersection. Assume light's incoming direction being Vi, the outgoing direction Vo would be

$$\boldsymbol{V_o = 2(V_i \cdot N)N - V_i}$$

And the fraction of reflected light is evaluated by Fresnel formula.

The E and F functions of BSDF of SpecularReflection returns 0; the G function calculates the outgoing direction and evaluates the fraction of reflected light using Fresnel equations.

Here's a sample image demonstrating specular reflection, with FresnelSpecial equation:

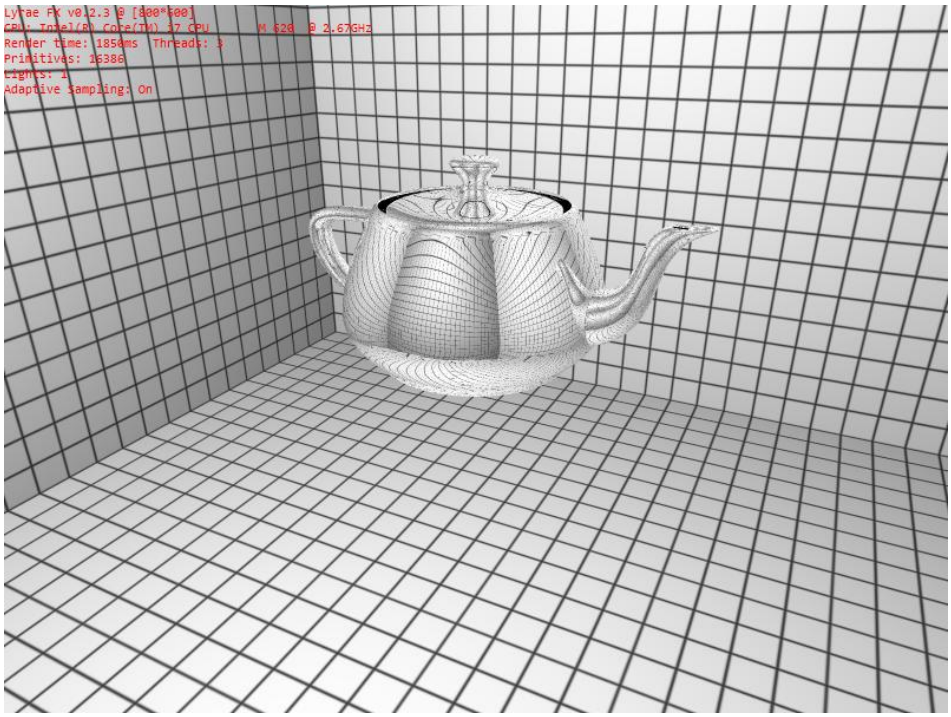

Figure 17 Specular Reflection

Specular Transmission (Refraction)

Specular transmission is very similar to specular reflection, with two differences: 1) the Fresnel type of specular transmission can be only FresnelDielectric; 2) the outgoing direction of transmitted light is calculated using Snell's Law:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

The G function calculates the direction of transmitted light using Snell's Law, and evaluates the fraction of light using Fresnel equations.

Here's an image demonstrating specular transmission (refraction), with refractive index set to 1.5 (similar to glass material):

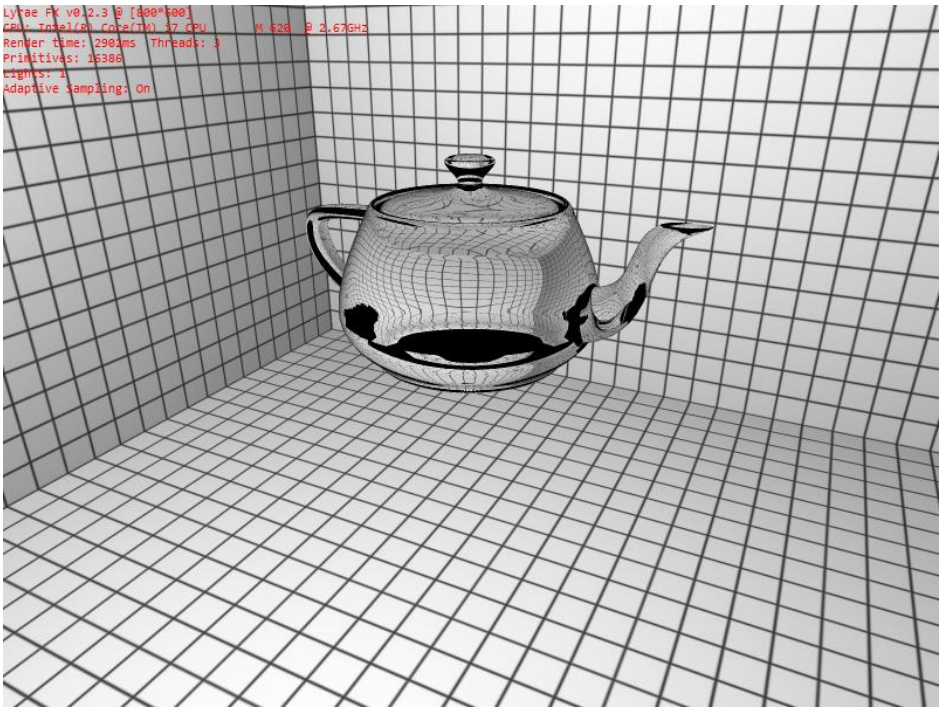

Figure 18 Specular Transmission (Refraction)

Glossy Reflection

Glossy reflection describes reflections at imperfect surfaces. This kind of behavior can be seen on materials such as mat glass. The basic idea of glossy reflection implementation is by shooting N rays at reflection point instead of only one ray, and getting N samples to evaluate the final color. These N rays should be random, and have a small offset with the perfectly reflected ray. This kind of method is called Monte Carlo method, and will be mentioned again in the Monte Carlo Method and Distributed Ray Tracing section.

Here's an image demonstrating glossy reflection, with glossy scale = 0.3 and samples = 32:

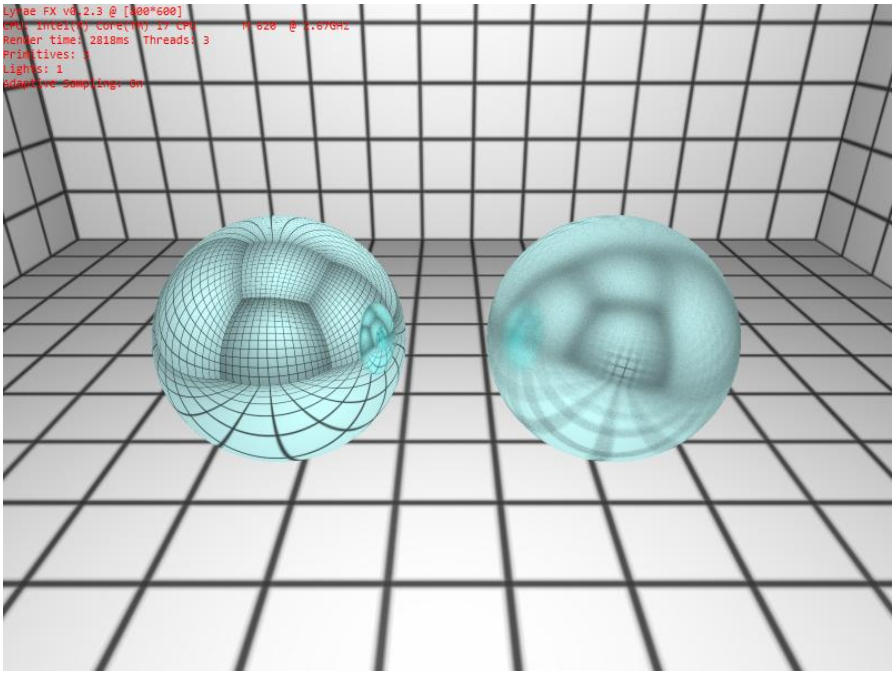

Figure 19 Glossy Reflection, scale=0.3, 32 samples

Glossy Transmission

Similar to glossy reflection, glossy transmission takes the similar steps by shooting N rays simultaneously. We'll omit the details here because most of it is the same as glossy reflection. Here's an image demonstrating glossy transmission, with glossy scale = 0.15 and samples = 32:

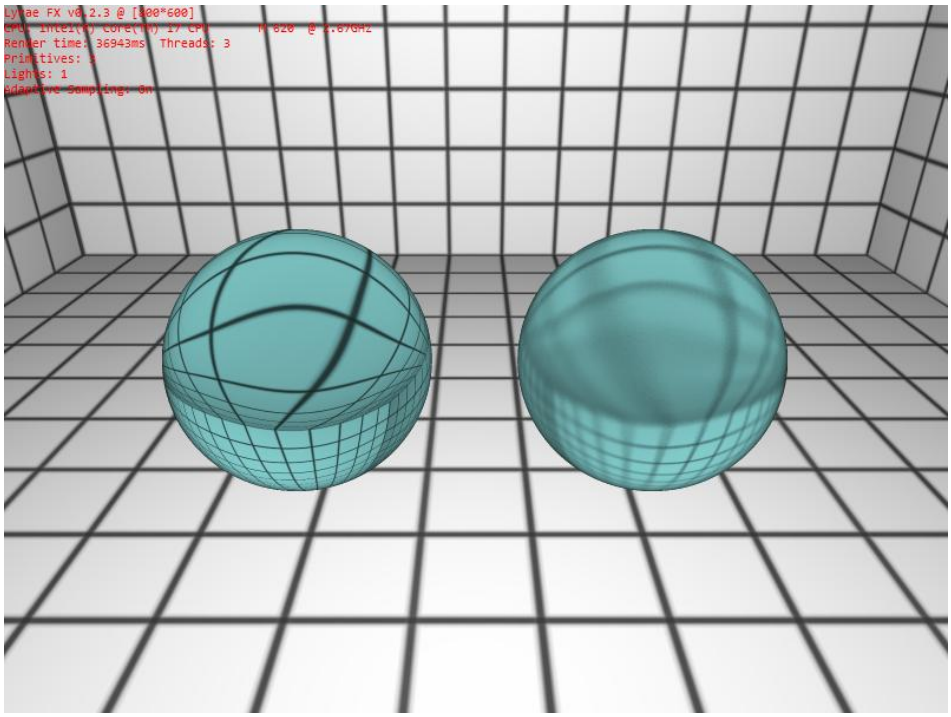

Figure 20 Glossy Transmission, Scale=0.15, 32 samples

Real World Materials

The BSDFs mentioned above are just a small subset of real world materials, but we can use these BSDFs to not-so-perfectly simulate real materials. Often, materials in Lyrae FX consist of several BSDFs, because some of the behaviors may coexist. For example, the typical glass material contains a SpecularReflection and a SpecularTransmission, for reflection and transmission take place at the same time.

Figure 21 Chessboard with glass material

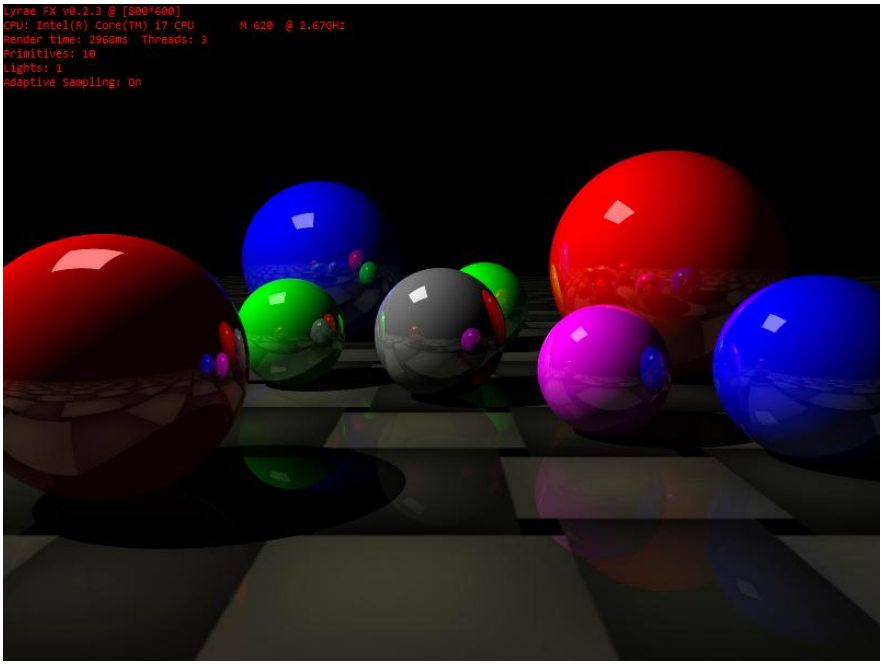

Figure 22 Balls' material consisting of Phong and SpecularReflection

There's also a large kind of material demonstrating the behavior of subsurface scattering, which light penetrates the surface of a translucent object, is scattered by interacting with the

material, and exits the surface at a different point. These materials are usually described by B Scattering Surface RDF (BSSRDF), which is not yet implemented by Lyrae FX.

5. Camera Models

Lyrae FX supports two kinds of camera models: perspective and orthographic.

Orthographic camera

Orthographic camera directly projects the sampled scene into the projection plane, and it preserves relative distance between objects. However, orthographic camera doesn't give the effect of foreshortening, which makes object smaller as they get farther away.
The image below demonstrate orthographic camera:

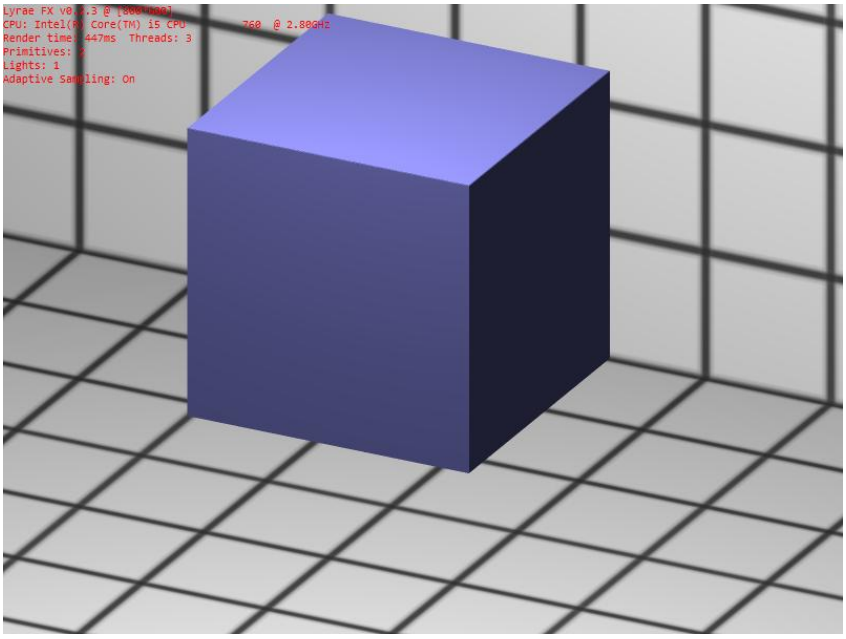

Figure 23 Box rendered by orthographic camera model

Perspective camera

Perspective camera is more like what we see by our eyes, as it includes the effect of foreshortening. Perspective camera shoot rays from one point, thus farther objects appear smaller than nearer objects.
There are two main parameters that determine the behavior of perspective camera, EyePosX and ProjPlaneHalfWidth, ProjPlaneHalfHeight. The ratio of ProjPlaneHalfWidth and ProjPlaneHalfHeight determines the aspect ratio of the perspective camera, and adding EyePosX, the camera's field of view (FOV) can be easily calculated.
The image below demonstrates perspective camera:

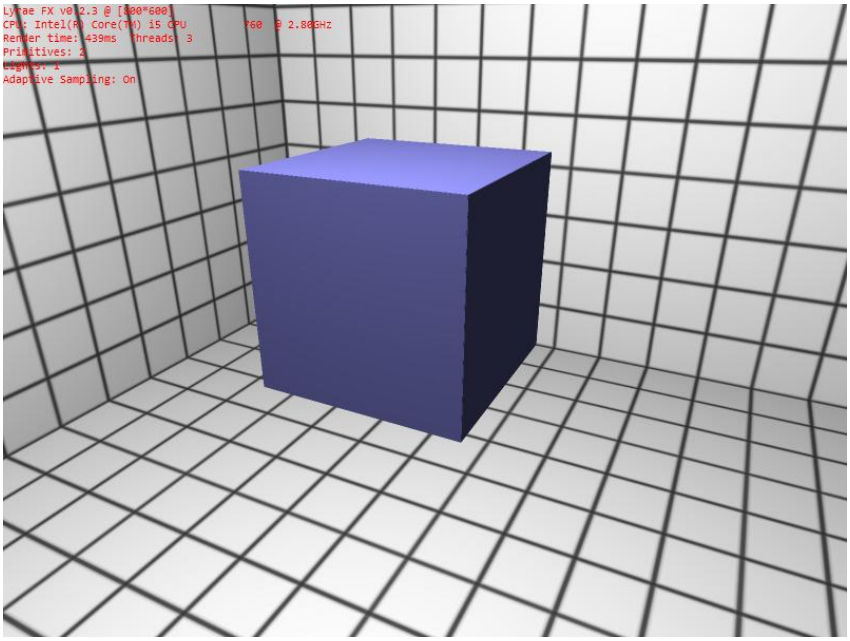

Figure 24 Box rendered by perspective camera model

6. Sampling

Sampling is an important process in rendering, which substantially determines the final image quality. Sampling defines how we retrieve the color information from the scene. Basically, if we get one sample per pixel, the synthesized image contains serious aliasing. The image below shows this situation.

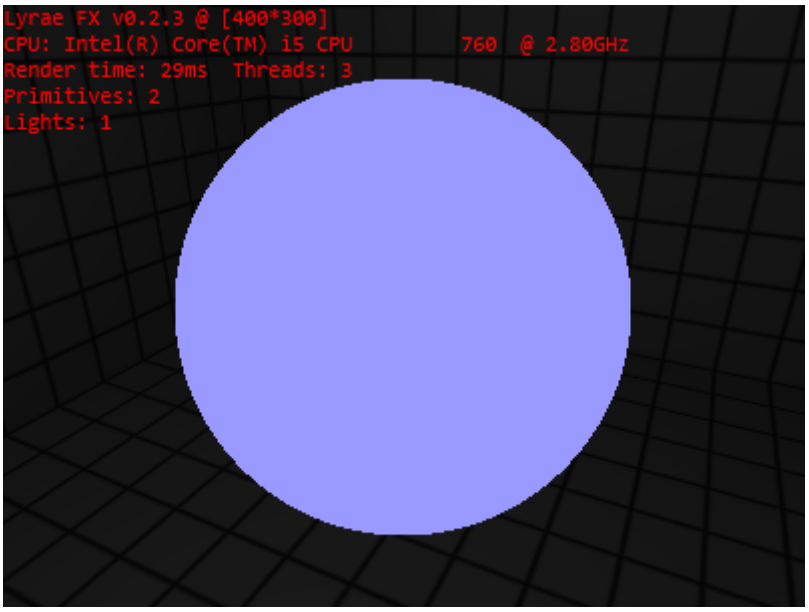

Figure 25 Aliasing: one sample per pixel, the image contains aliasing, giving a bad visual experience

To solve this deficiency, Lyrae FX supports two kinds of anti-aliasing methods: Supersampling and Adaptive supersampling.

Supersampling

The basic idea of Supersampling is to take several samples per pixel, and then average these values to get the final image. With high sampling rate, this method can get very high quality images, with totally smooth shape edges. The image below shows supersampling.

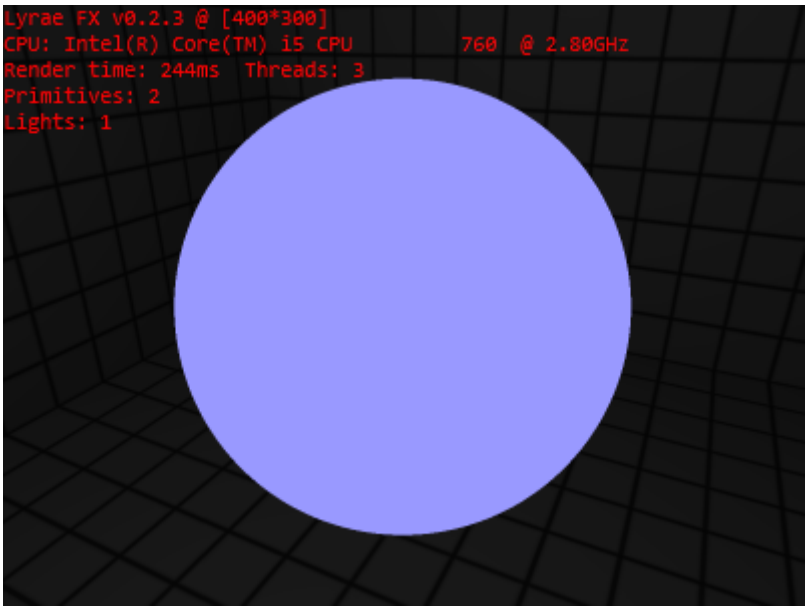

Figure 26 Antialiasing by supersampling, the edge becomes much smoother than the first image

One problem about supersampling is that its efficiency can be really low when more samples are taken, because most of the image doesn't need supersampling at all. To solve this problem, Lyrae FX supports another sampling method: adaptive supersampling.

Adaptive supersampling

Adaptive supersampling is based on the idea that only the pixels that contain large difference of contrast should get more samples. Shape's edges, shadows, and other things with clear outline are easy to cause aliasing, because the sudden change of contrast makes the image deficiency very obvious.

To perform adaptive supersampling, an initial set of a few (say, four) samples of a pixel is taken. Then, the luminance of each sample is computed, also their average luminance. After that, if the difference between the average luminance and any sample's luminance is larger than a certain threshold, this pixel should be sampled again, with more samples (say, sixteen). In this way, only those pixels with large contrast difference will be taken more samples, thus may increase both the performance and image quality.

The image below is rendered with adaptive supersampling, an initial of 4 samples per pixel is taken, if supersampling is needed, a larger set of 16 samples is taken again.

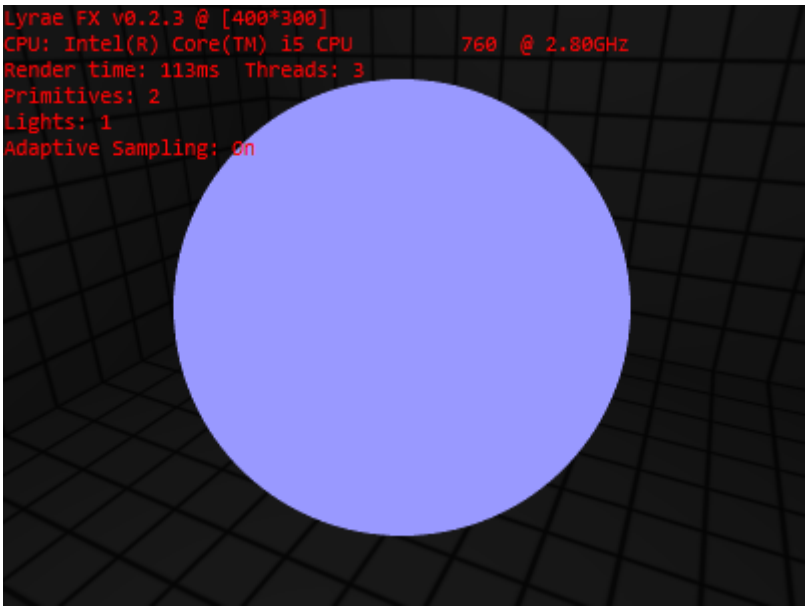

Figure 27 Adaptive supersampling

Here's a comparison of these sampling methods:

Method	None	Supersampling	Adaptive supersampling
Time used	29ms	244ms	113ms
Image quality	Bad	Good	Good

Importance Sampling

A refinement of Monte Carlo method (discussed later) is importance sampling. Importance sampling in statistics is to somehow make the points random, but more likely to come from regions of high contribution to the integral than from regions of low contribution.

A recursive ray tracer spawns secondary rays at each intersection point. In case of soft shadows and diffuse reflections, this can result in enormous amounts of rays. This situation can be improved using importance sampling. For the first intersection, it's still a good idea to spawn lots of secondary rays to guarantee noise-free soft shadows and high quality diffuse reflections. The secondary rays however could do with far less samples: A soft shadow reflected by a diffuse surface will probably look the same when we take only a couple of samples.

Lyrae FX supports importance sampling of glossy reflection/transmission, depth of field, soft shadows and other situations where multiple samples are taken.

7. Lights

Lyrae FX currently supports three kind of light sources: point light, area light and spot light. Lights themselves are not visible, but they can light other objects, making the whole scene visible to observers.

Basic light interface contains three attributes: **Color**, **Intensity** and **Attenuation (Alpha).** Color describes the light's color, and Intensity describes how strongly the light is emitting. Attenuation describe the attenuating rate that the energy of light decreases along the ray.

The **Sample()** member function is used to sample the color of light distributed at a given position. Each kind of light handles its own **Sample()** method.

Physically, the total light energy **d*E*** deposited on area **d*A*** is described by the following equation:

$$\mathrm{dE} = \frac{\Phi \cos\theta}{4\pi r^2}$$

In Lyrae FX, the equation above is modified to:

$$\mathrm{dE} = \frac{\Phi \cos\theta}{4\pi[1 + (\frac{r}{\alpha})^2]}$$

The equation above can prevent too strong light intensity at short distances, and gives control to the light's attenuation.

The $\cos\theta$ term is evaluated is BSDF's **F()** member function, and the $[1 + (\frac{r}{\alpha})^2]$ term is evaluated in Light's **Sample()** member function.

Point Light

Point lights are a common kind of light which is abstracted to a point in geometry. They emit energy equally to all directions.

The Sample() method of point light calculates the distance between light and target, and evaluates the color using the equation above.

Here's a scene containing two point lights. Note that they create distinct shadows.

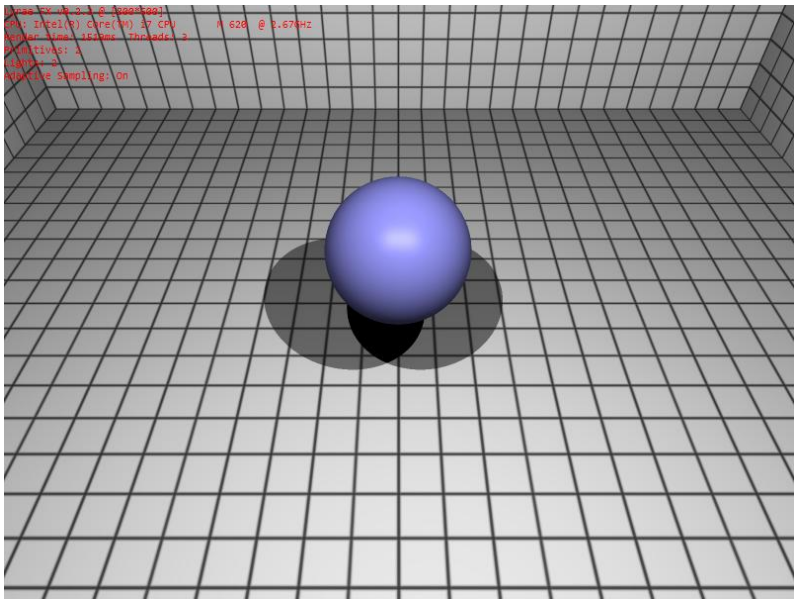

Figure 28 Scene containing two point lights

Area Light

The difference between area light and point light is that area lights have volume larger than zero. Current implementation of area lights in Lyrae FX represents them as AABBs for simplicity. The Sample() method for area lights is similar to the one of point lights. The position of area light is represented by the centroid of AABB.

Because area lights have non-zero volume, they create shadows with blurred edges, which is called Soft Shadow. The implementation of soft shadows is described in the Distributed Ray Tracing section. The following image demonstrates a scene with an area light, creating soft shadow:

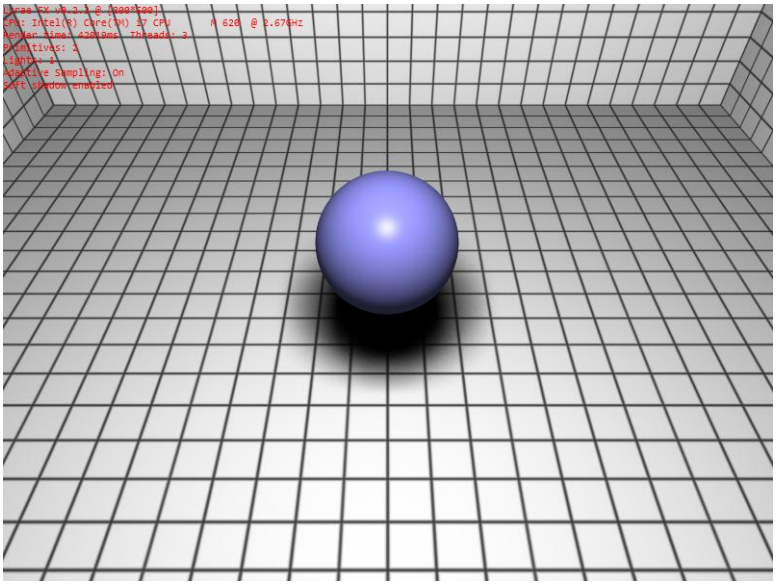

Figure 29 Soft shadow created by area light

Spot Light

Spot lights are the kind of light which emit light within a certain angle. Spot light has two additional attributes: Alpha0 and Alpha1. When the angle between viewer's direction and light's emitting direction is less than Alpha0, the strength of light is 100%; from Alpha0 to Alpha1, the strength degrades from 100% to 0%; and if the angle is larger than Alpha1, the strength is 0%. Therefore, in the Sample() function of spot light, it calculates the angle and adjust the light's strength.
Here's an image demonstrating spot light:

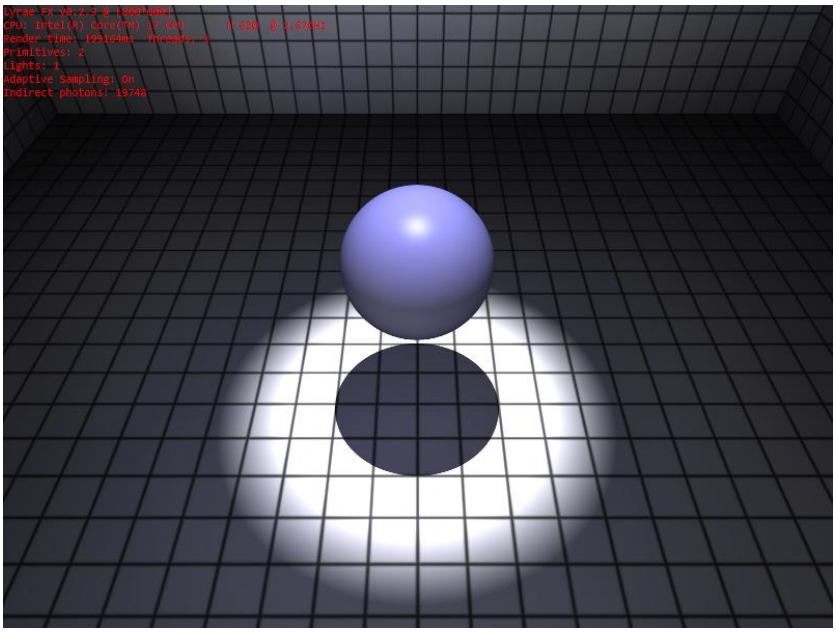

Figure 30 Spot light

Shadows

Shadows are created by shooting rays from sampling point to light sources, and if the resulting intersection is nearer than the light's distance, this light source is shaded thus no light can reach the sampling point from that light source.
For soft shadows, however, many samples of the area light are taken to evaluate how much the light source is shaded. This will be introduced in the Distributed Ray Tracing section.

8. Texture

Applying texture to shapes is a simple yet very effective idea to improve the degree of realism of rendered image. Basically, texture is just a 2D image. Each point of shapes has a texture coordinate (UV coordinate) specifying the texture position to sample the texture.

Texture Mapping

Sphere

Lyrae FX currently supports the simple approach of spherical textures, which directly maps latitude-longitude onto sphere. The texture source use a rectangular texture with proportion of 2x1, and the U coordinate goes from 0 to 1 around the equator, v goes from 0 to 1 from pole to pole.

Spherical texture mapping is mainly used to do environment mapping in Lyrae FX. By creating a large sphere which contains the entire scene, an environment map is applied to that sphere, and we can get pretty good result. The following image shows this kind of texture mapping.

Figure 31 Spherical texture mapping used as environment mapping

Box

Texture mapping for boxes is simpler than sphere. The UV coordinates are directly calculated according to the proportion of that point on the surface. The image below shows box texture mapping.

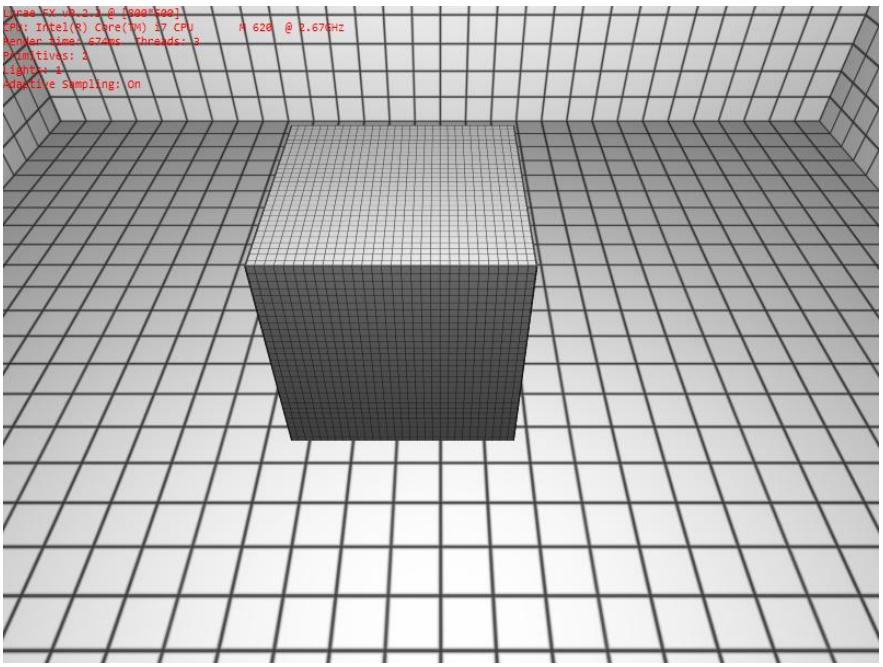

Figure 32 Box texture mapping

TriangleMesh

The UV coordinate for triangle mesh is pre-defined and loaded when loading the mesh. Each vertex is assigned with a pair of UV coordinates. When calculating the UV coordinate of any arbitrary point on the mesh, barycentric coordinate is used again. The final UV coordinate would be averaging the UV coordinate of each vertex. The image below shows texture mapping for triangle mesh:

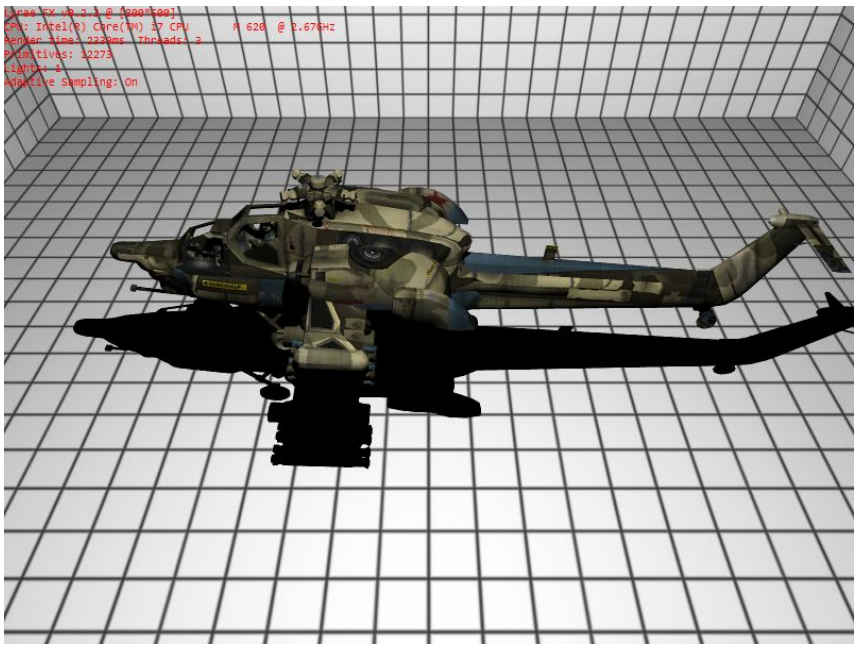

Figure 33 Mesh texture mapping

Texture Filtering

Because the UV coordinates are in float format and textures are discrete image points, directly mapping UV range (0, 1) to texture width (height) may produce unpleasant texture quality. To improve texture quality, texture filtering should be applied.

Lyrae FX currently supports bilinear texture filtering. Bilinear filtering uses four texels (texture pixel) instead of one for each sample to perform bilinear interpolation between the four texels nearest to the point that the pixel represents.

The pseudo code for bilinear filtering is shown below:

```
double getBilinearFilteredPixelColor(Texture tex, double u, double v)
{
  u *= tex.size;
  v *= tex.size;
  int x = floor(u);
  int y = floor(v);
  double u_ratio = u - x;
  double v_ratio = v - y;
  double u_opposite = 1 - u_ratio;
  double v_opposite = 1 - v_ratio;
  double result = (tex[x][y]   * u_opposite  + tex[x+1][y]   * u_ratio) * v_opposite +
                  (tex[x][y+1] * u_opposite  + tex[x+1][y+1] * u_ratio) * v_ratio;
  return result;
}
```

9. Monte Carlo Methods and Distributed Ray Tracing

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Lyrae FX uses Monte Carlo methods to generate random walks through the scene, where the pixel intensities are estimated by making the average of their contributions. Lyrae FX achieves advanced effects such as soft shadow, depth of field and glossy reflection using Monte Carlo methods.

Basically, Lyrae FX traces many but finite number of samples to evaluate an integral value of one single point. Each sample is traced individually, and their average value is used to construct the final result. In the following sections, some of the effects using Monte Carlo methods are introduced.

This ray tracing method using Monte Carlo methods is called **distributed ray tracing**.

Soft Shadow

Soft shadows are produced when an area light is used to light up the scene. Comparing to point lights, area lights create blurry shadow edges. The example below demonstrates soft shadow:

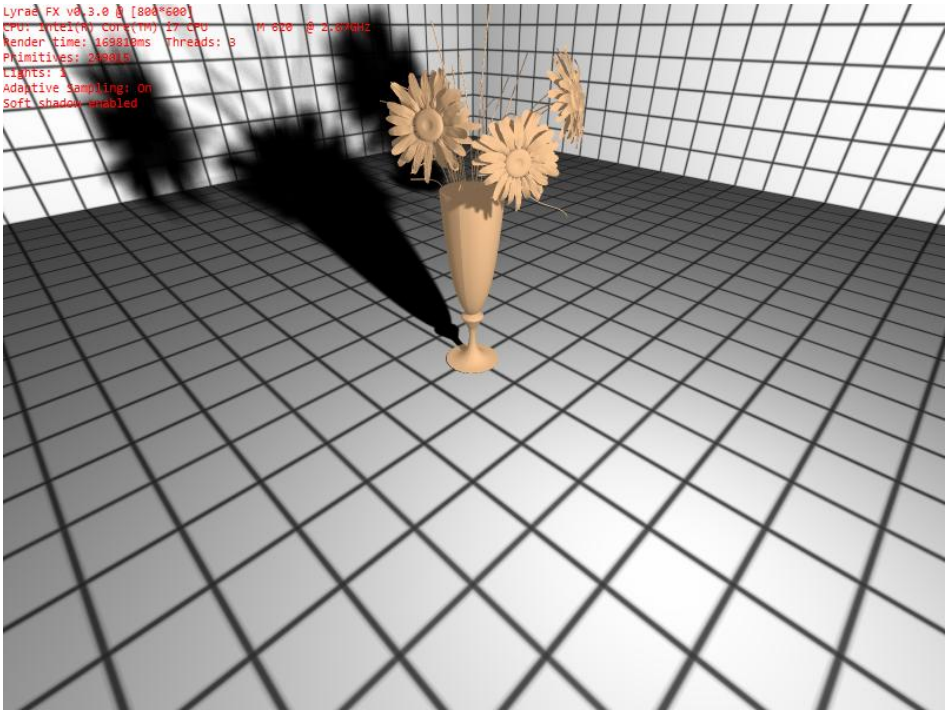

Figure 34 Rendered with soft shadow enabled

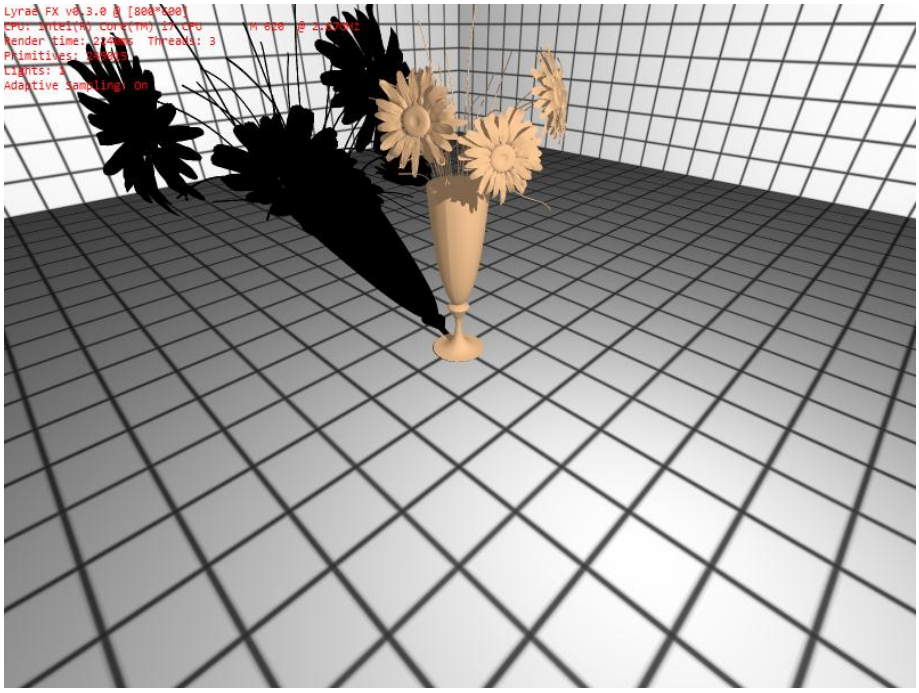

Figure 35 Rendered with soft shadow disabled

Soft shadow is created by tracing multiple rays to judge if they intersect with the light source. Each light source sampling position is a random position inside the AreaLight's bounding box. (AreaLight uses an AABB to represent its geometry for simplicity). The ratio of samples that create shadow is divided by the total samples, and this result is used to evaluate the final shadow value. Too few samples will produce much noise, so normally 128 samples is a reasonable choice. Rendering with soft shadow is quite slow, for each pixel is judged many (128) more times of light intersection.

Depth of Field

In optics, the depth of field is the portion of a scene that appears sharp in the image. Both the human eye and cameras have a finite lens aperture, and therefore have a finite depth of field. Lyrae FX creates depth of field by placing an artificial lens in front of the view plane. Randomly distributed rays are used to simulate the blurring of depth of field. The first ray cast is not modified by the lens. It is assumed that the focal point of the lens is at a fixed distance along this ray. The rest of the rays sent out for the same pixel will be scattered about the surface of the lens. At the point of the lens they will be bent to pass through the focal point. Points in the scene that are close to the focal point of the lens will be in sharp focus. Points closer or further away will be blurred. As illustrated by the following graph:

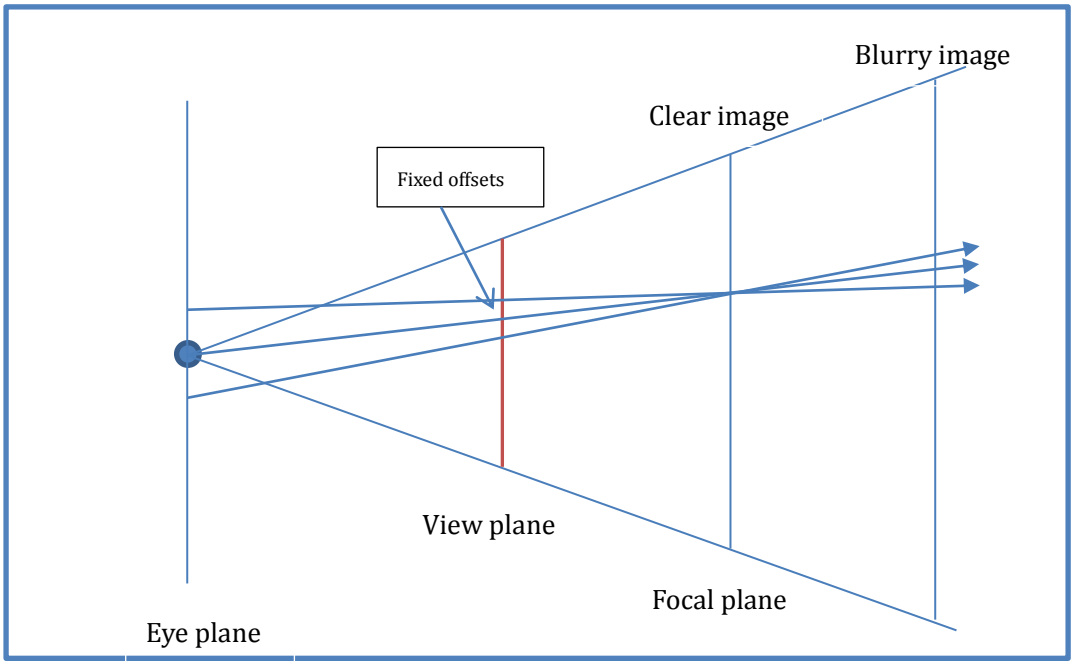

Figure 36 Depth of Field illustrated

The following example demonstrates Depth of Field effect:

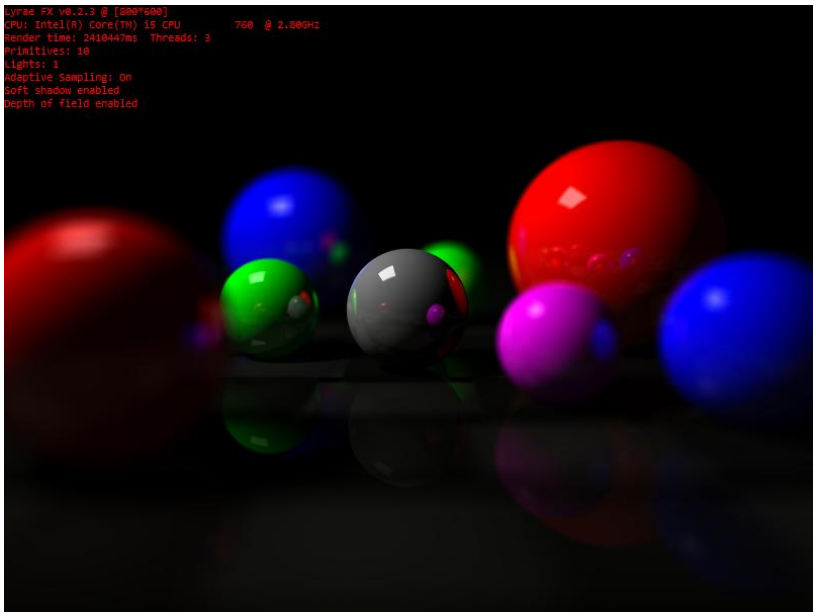

Figure 37 Rendered with Depth of Field enabled

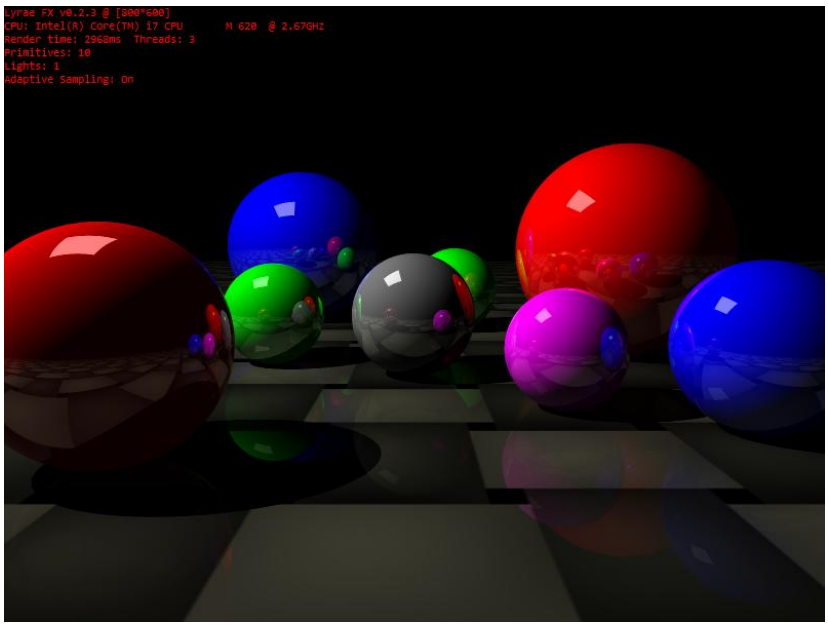

Figure 38 Rendered with Depth of Field disabled

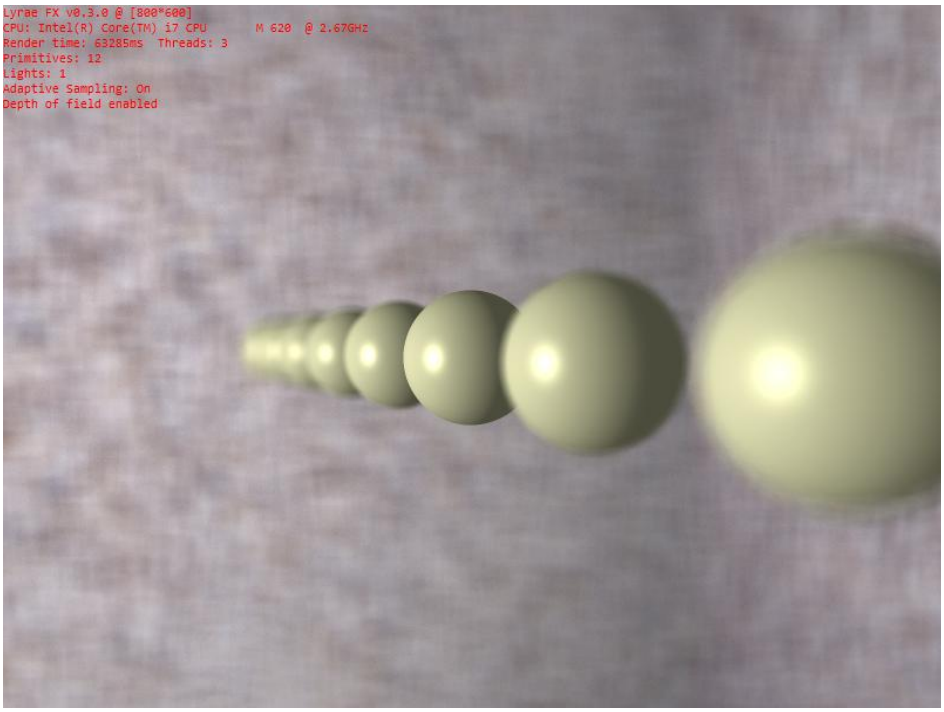

Figure 39 Another scene rendered using Depth of Field

Glossy Reflection and Transmission

Glossy reflection and transmission are also rendered using Monte Carlo methods. When perfect reflection or transmission is rendered, only one ray is traced to evaluate the reflected or transmitted result. In distributed ray tracing, however, several rays within a certain angle of the perfect ray are traced together to evaluate the final result. Typically, 64 samples for glossy reflection and 16 samples for glossy transmission are a minimum number of samples required to reduce the noise to a reasonable range.

The samples are generated by BSDF introduced in earlier section. A glossy factor is used to determine the range of random directions.
Here are two images showing glossy reflection and glossy transmission.

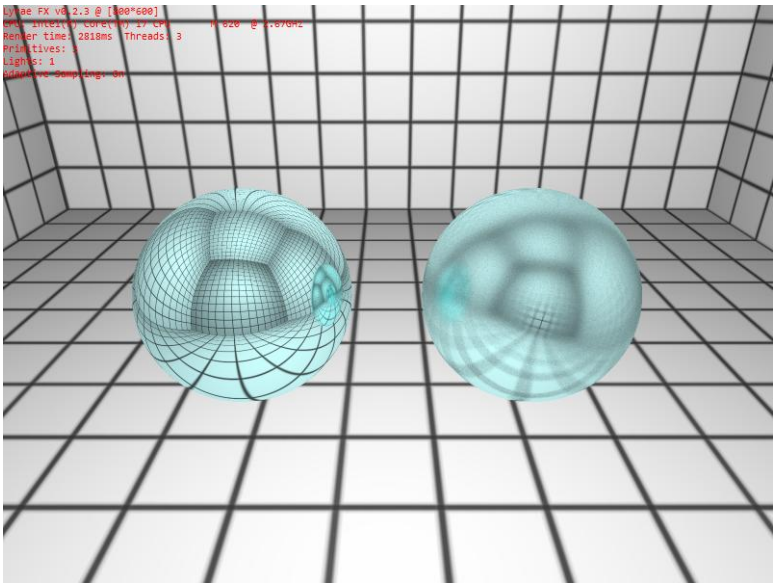

Figure 40 Perfect/Glossy reflection

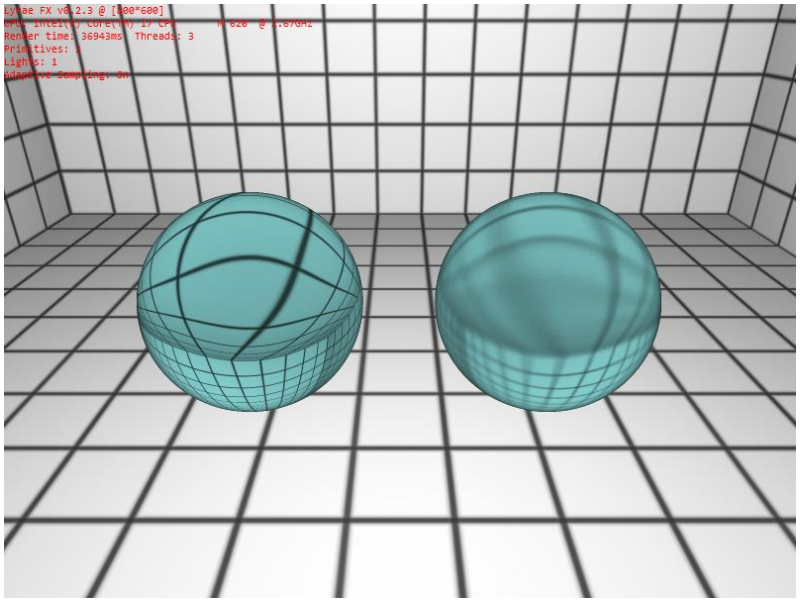

Figure 41 Perfect/Glossy transmission

Other distributed ray tracing effects such as motion blur are not yet implemented in Lyrae FX.

10. Global Illumination

Global illumination is a general name for a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light which comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflection or non (indirect illumination). In practice, only the simulation of diffuse inter-reflection or caustics is called global illumination.
Lyrae FX currently takes three separate approaches towards global illumination. They are **Instant Global Illumination**, **Ambient Occlusion** and **Photon Mapping**.

Instant Global Illumination (IGI)

Instant Global Illumination is a simple global illumination method that shoots lights particles into the scene, and places Virtual Point Lights where these light particles hit the scene. The scene is then ray traced, and illumination is computed by shooting shadow rays towards these virtual lights.
Lyrae FX implements IGI in a rather simple way. Before the rendering process, light sources shoot random rays into the scene. Upon each intersection, a virtual light is created, locating near the intersection point. Then, the scene is rendered in the traditional way, where virtual lights are evaluated as well as the actual lights.
This image below demonstrates an earlier work of implementation of IGI:

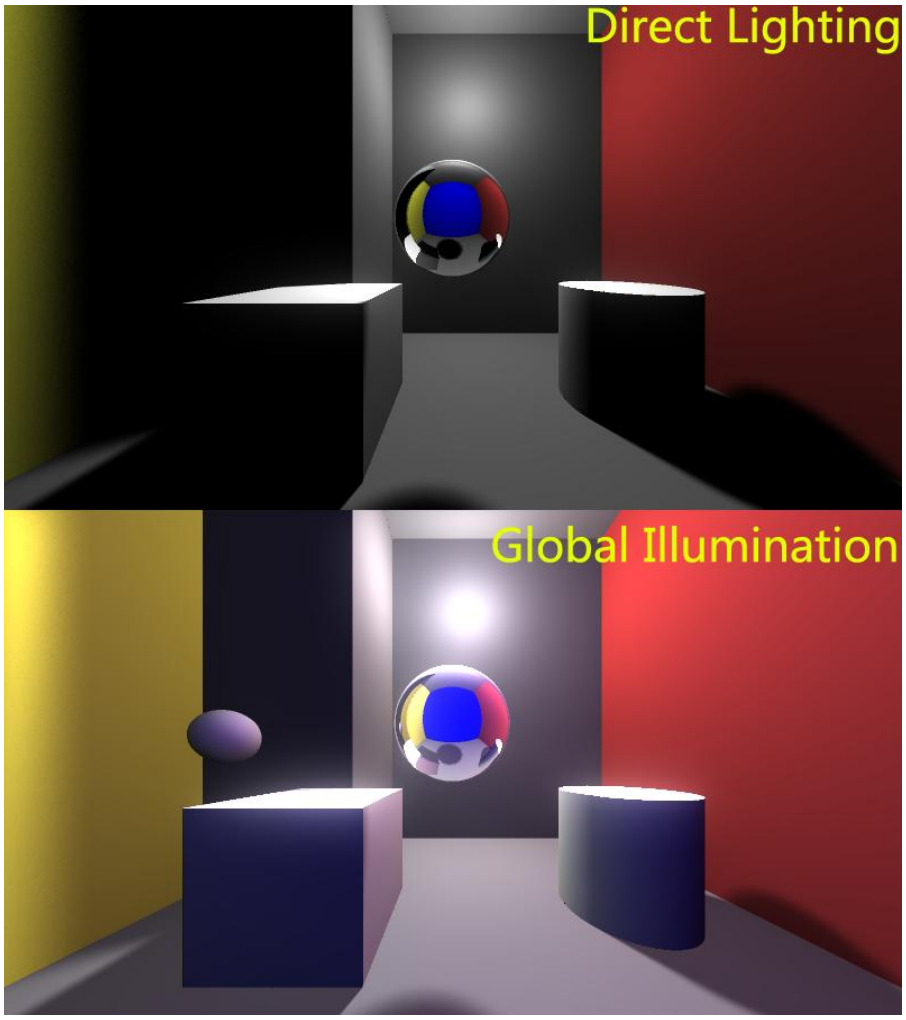

Figure 42 Direct lighting vs Instant Global Illumination

From the image above, we can see that the shadowed area in the second image is also visible because of the virtual lights.

Lyrae FX's IGI implementation is currently rather naïve, because it's difficult to produce precise indirect illumination. GI in Lyrae FX is better using Ambient Occlusion and Photon Mapping, as illustrated below.

Ambient Occlusion

Ambient Occlusion is a more advanced shading method which helps add realism to local reflection models by taking into account attenuation of light due to occlusion.

Ambient occlusion is most often calculated by casting rays in every direction from the surface. Rays which reach the background or "sky" increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry are rendered dark, whereas points with little geometry on the visible hemisphere appear light.

Ambient Occlusion is used most often in an environment filled with ambient light, such as a in a sky sphere where there's no significant direct lighting. Upon the rendering of each pixel, some random rays are shoot to evaluate how 'occluded' a position is. The lighting of the

position is then evaluated accordingly.

Images below demonstrate Ambient Occlusion.

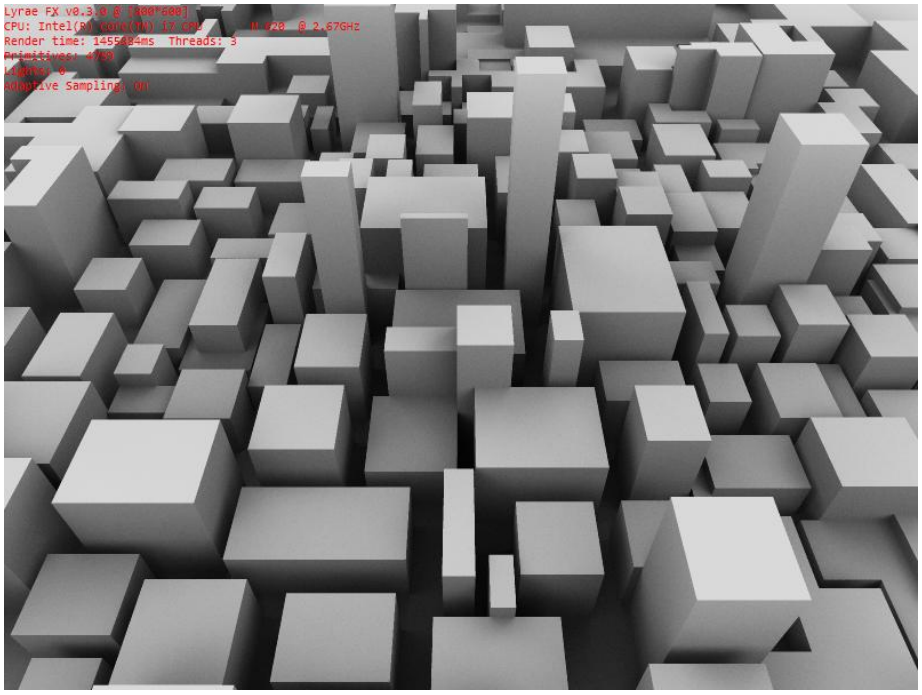

Figure 43 Urban scene rendered using Ambient Occlusion

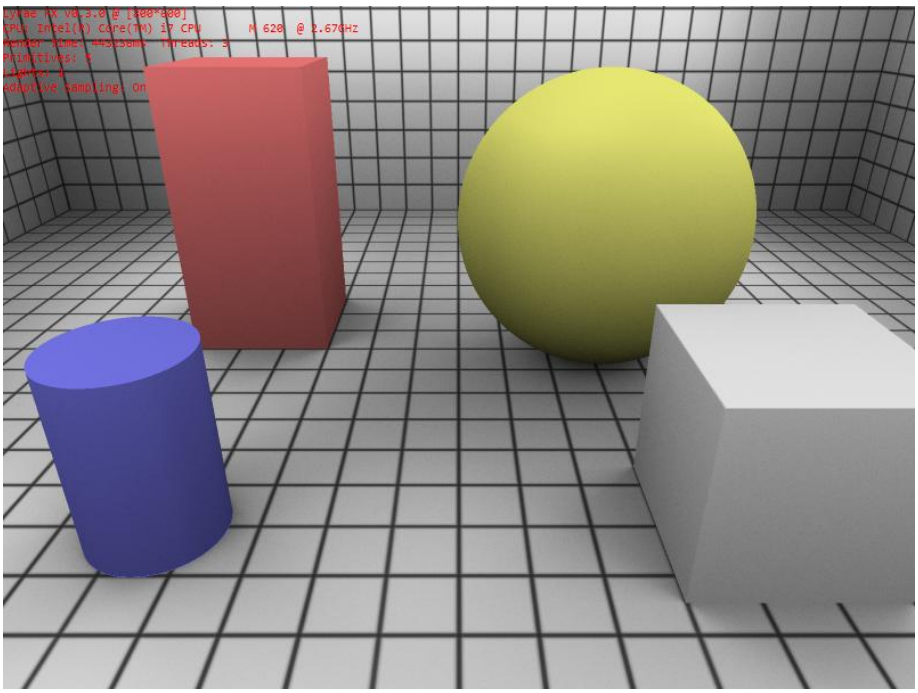

Figure 44 Primitives rendered using Ambient Occlusion

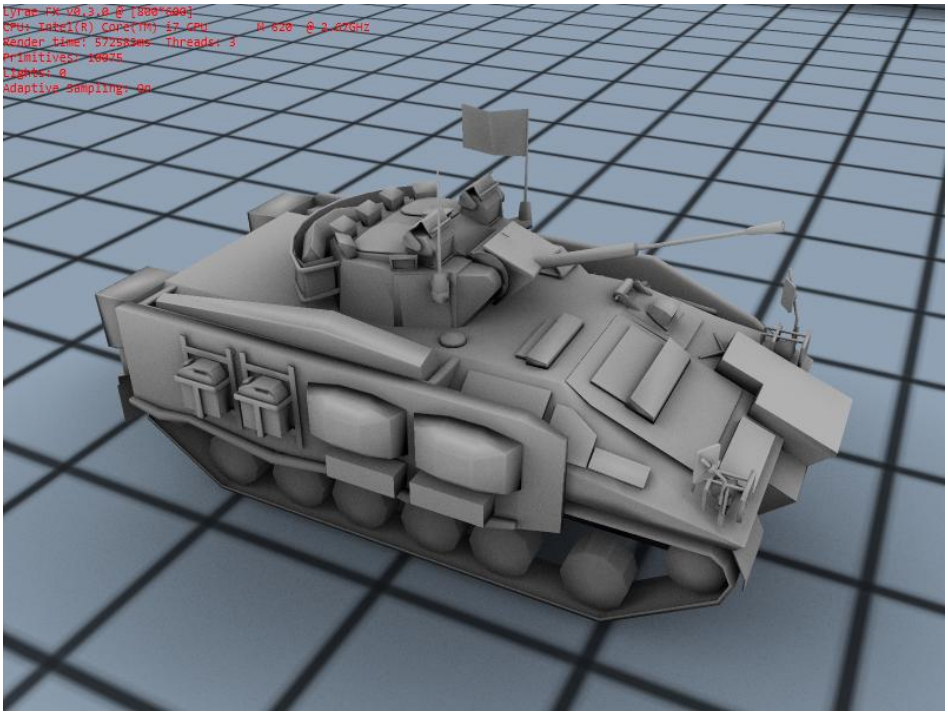

Figure 45 Tank model rendered using Ambient Occlusion

Photon Mapping

Photon Mapping is a two-pass global illumination algorithm that solves the rendering equation. It is used to realistically simulate the interaction of light with different objects. Specifically, it is capable of simulating the refraction of light through a transparent substance such as glass, and diffuse inter-reflection between illuminated object.

There are two passes in the photon mapping algorithm. They are:

(1) **Construction of the photon map**: Light packets called photons are sent out into the scene from the light sources. Whenever a photon intersects with a surface, the intersection point and incoming direction are stored in a cache called the **photon map**. Two photon maps are created in Lyrae FX: one especially for caustics and a global one for indirect illumination, because typically rendering caustics accurately requires more photons.

The 1st pass is subdivided into 3 parts:

(a) **Photon Emission**: A photon begins at the light source. For each light source a set of photons are created according to the power (intensity) of the light source. Current implementation of photon emission in Lyrae FX has two shooting policy: one is to shoot the photons uniformly into all directions, and alternatively, photons are shoot into a directed target (specified by an AABB), which is called directed photon shooting. For caustics photons, each primitive which has a BSDF flag of reflection or transmission is recorded, and one of them is chosen randomly. Then, photons are shoot towards the bounding box of that primitive.

(b) **Photon Scattering**: Emitted photons from light sources are scattered through a scene and are eventually absorbed or lost. The scattering policy for indirect and caustics photons are different: indirect photons are scattered into a random direction when it hits a surface, when the surface has BSDF flag of diffuse; for other kind of surfaces, the photon

stops emitting. And for caustics photons, only the surfaces that have flags of specular reflection or specular transmission are set to scatter these caustics photons. For both scattering policy, a maximum depth and an absorption probability are set. When a photon reaches the maximum depth or is evaluated to be absorbed, it simply disappears. This simplified photon scattering policy can improve the performance of Lyrae FX's photon mapping.

These two parts is implemented in PhotonMap::ShootIndirectPhoton() / ShootCausticsPhoton().

(c) **Photon Storing**: For a scene containing millions of photons, efficient photon selection is required. Lyrae FX uses kd-tree to store photons. The data structure of kd-tree will not be described here, the only thing we should know is that it provides fast range selection in three dimensional space. The photon's position, as well as it incoming direction and color (radiance) is stored in the kd-tree.

(2) **Rendering**: In this pass of the algorithm, the photon map is used to evaluate the radiance of every pixel of the image. The rendering strategies for indirect and caustics photons are basically the same.

During the rendering pass we are using an approximation of the rendering equation to compute the reflected radiance at surface locations. The rendering equation in the form that is typically used is described as:

$$L_r(x, \vec{\omega}) = \int_{\Omega} f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}) \cos\theta_i d\vec{\omega}$$

By averaging the overall radiance to separate photons, this radiance value is evaluated as:

$$L_r(x, \vec{\omega}) = \sum_{p=1}^{n} f_r\left(x, \vec{\omega}_p, \vec{\omega}\right) \frac{\Delta\Phi_p(x, \vec{\omega})}{\pi r^2}$$

Basically, when rendering photons, we choose the N nearest photons to evaluate the total radiance. Different Ns and selection ranges are chosen for indirect and caustics photons rendering. A large range for indirect selection is chosen to get a smooth illumination, and a small range for caustics selection is chosen to accurately render the caustics. This is reasonable because there're typically much more photons of caustics than indirect, where the former is usually 10^6 and the latter is usually 10^4.

Photon rendering is implemented in PhotonMap::EvaluateIndirectRadiance() / EvaluateCausticsRadiance().

Another process called final gathering can be used to increase the image quality. However, final gathering is quite time-consuming, which significantly increase the rendering time. Current version of Lyrae FX does not support final gathering.

Indirect illumination is illustrated by the following images:

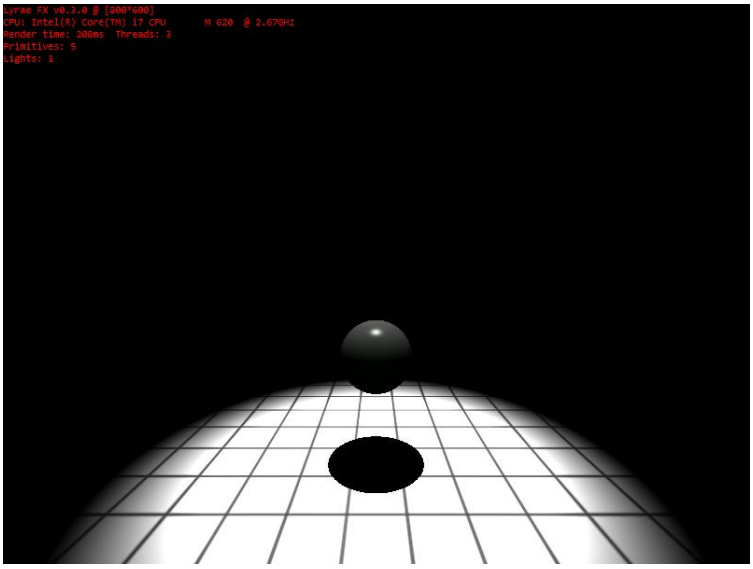

Figure 46 Image rendered without global illumination as a reference

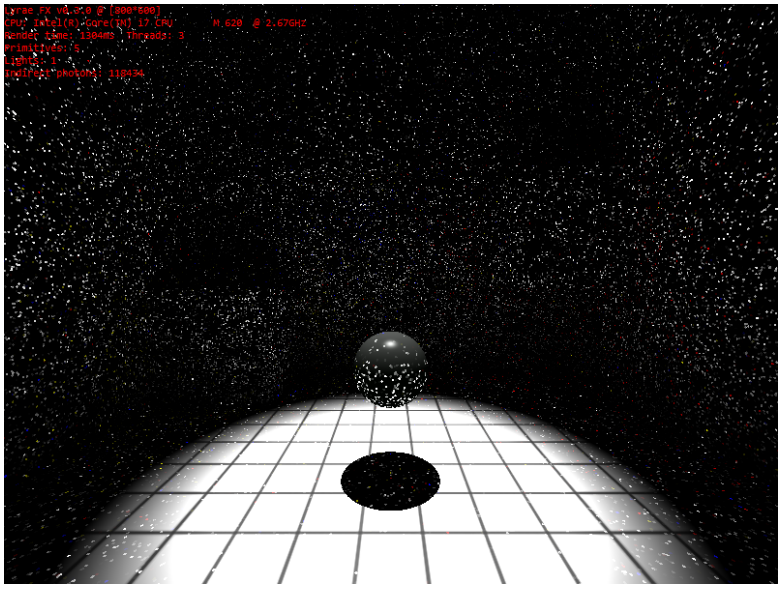

Figure 47 The indirect photon map

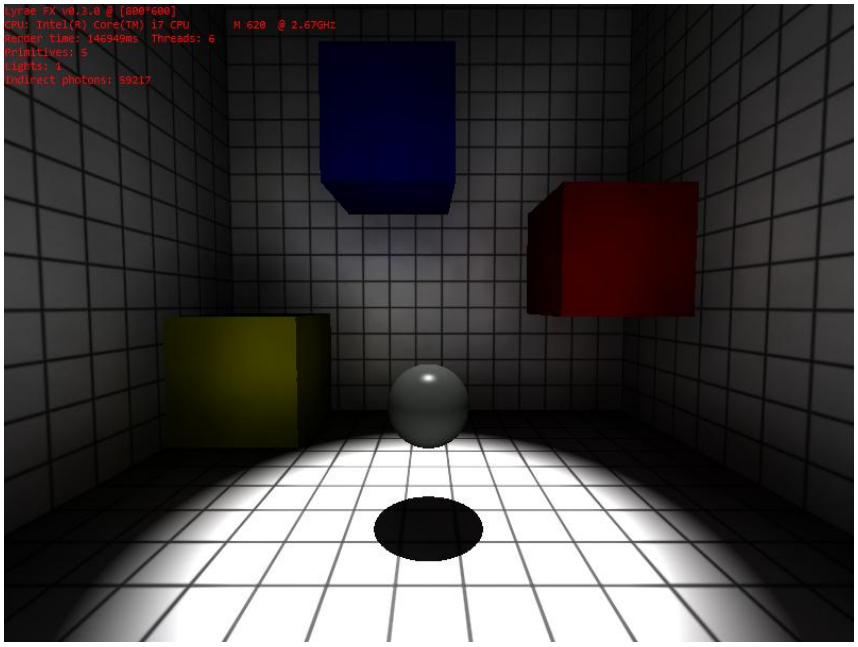

Figure 48 GI using photon mapping, see the color blending on the wall

Another use of photon mapping is the rendering of caustics. A caustic is the envelope of light rays reflected or refracted by a curved surface of object, or the projection of that envelope of rays on another surface. Lyrae FX supports the rendering of high-quality caustics.
Here's some images demonstrating the rendering of caustics.

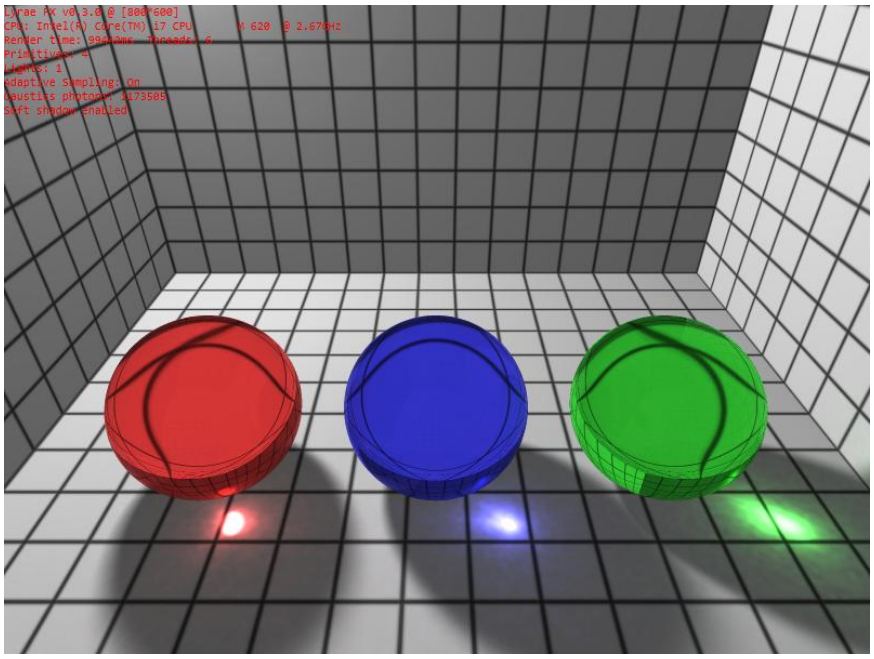

Figure 49 Image demonstrating refraction caustics

Figure 50 Image demonstrating reflection caustics

Figure 51 Tableware rendered with caustics

11. Volumetric Lighting

Volumetric Lighting allows the viewer to see beams of light shining through the environment. In volumetric lighting, the light cone emitted by a light source is modeled as a transparent object and considered as a container of a 'volume'; as a result, light has the capability to give the effect of passing through an actual three dimensional medium that is inside its volume, giving a striking visual effect.

The image below demonstrates Volumetric Lighting:

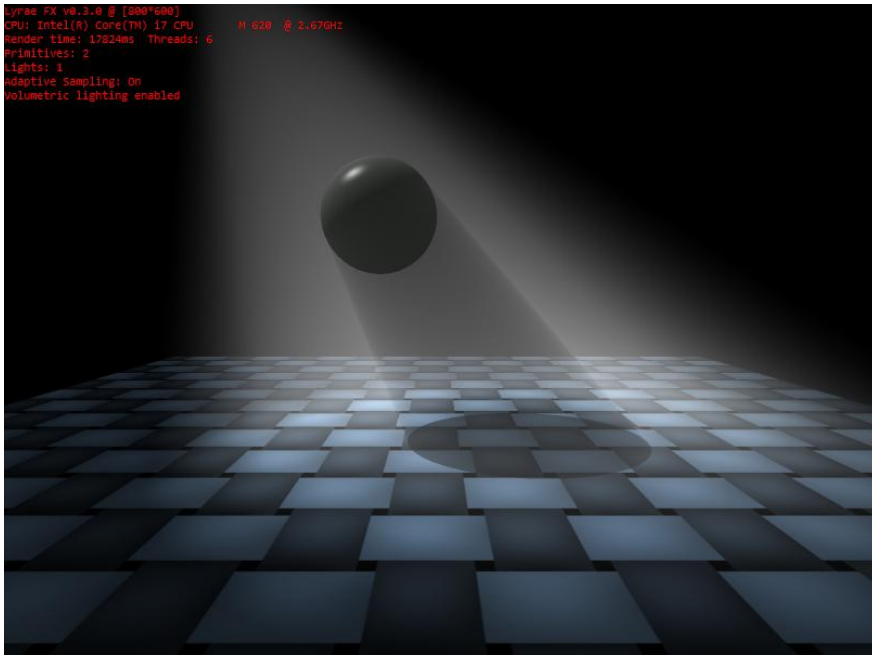

Figure 52 Volumetric lighting

The basic idea of volumetric lighting implementation is that for each traced ray from camera, samples of radiance along the ray are taken to evaluate how much radiance the ray can get from the volumetric environment. For each traced ray, a sampling step distance is first calculated. More accurate rendering can be gained by decreasing the sampling step. Then, a random offset of each ray is generated, and the first sampling position is the offset. After that, samples are taken at the same interval of the sampling step, until it reaches the maximum sampling distance. This process is illustrated as follows:

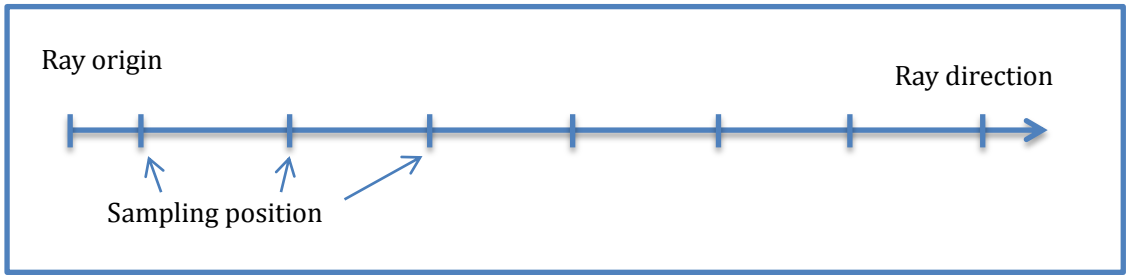

Figure 53 Volumetric lighting illustrated

The image below shows another scene rendered using volumetric lighting.

Figure 54 Tree rendered using Volumetric Lighting and Ambient Occlusion

Figure 55 Another scene rendered using volumetric lighting

12. Post Processing

Lyrae FX uses HDR format as internal image format. HDR (High Dynamic Range) is a set of techniques that allow a greater dynamic range of luminance between lightest and darkest areas of an image. Internally in Lyrae FX, each pixel contains three float values that can represent a large range of luminance. However, when the image is presented to screen or saved on disk, each pixel contain only three 8-bit values representing its R,G,B parts of the color. Direct conversion from float color format (HDR format) to RGB format leads to substantial information loss. The post processing is used here to perform HDR to RGB format conversion, which uses techniques called **Tone Mapping** to preserve information.

Tone Mapping is a technique used in CG to map one set of colors to another, often to approximate the appearance of high dynamic range images in a medium that has a more limited dynamic range. Essentially, tone mapping addresses the problem of strong contrast reduction from the scene values to the displayable range while preserving the image details and color appearance important to appreciate the original scene content.

The images below shows tone mapping process:

Figure 56 Without tone mapping, scene outside the window is barely distinguishable

Figure 57 With tone mapping, more details are presented than the previous one

The post processing stage is a 2D image processing procedure, so the implementation details are omitted here.

13. Parallelism

Lyrae FX is designed on a fully parallel basis. Parallel running support of Lyrae FX takes advantages of multiple cores of modern processors, dispatching rendering tasks on separate cores, thus maximizing the system's running efficiency.

The parallelism support is mainly achieved by a careful design. Each class and member function is designed to be reentrant by multiple threads. Some synchronization techniques such as critical sections and atomic operations are provided to protect shared resources.

Lyrae FX API integrates the multi-thread managing. The GUI part of renderer is able to call the API directly, without having to consider the multi-thread issues.

Currently two main procedures support parallel running: main rendering procedure and photon map creating. Other operations that consume very little are run single-threaded for simplicity.

The implementation of parallel part is in Core/Parallel.h(.c) and Core/LyraeAPI.h(.c).

14. Performance

One of Lyrae FX's most important goal is to ensure good performance. Lyrae FX uses many techniques to speed up the rendering process.

Lyrae FX uses BVH for spatial subdivisions. BVH is a fast spatial subdivision algorithm which ensures fast rendering of scenes containing hundreds of thousands of triangles. Currently BVH is the only supporting spatial subdivision algorithm in Lyrae FX.

The parallel running of rendering process maximize usage of CPU. The graph below shows the running time of different threads used. (Scene: bowl, 6k triangles; CPU: Intel Core i7-620M, 2 Cores, 4 Threads; OS: Windows 7 Ultimate 64-bit; low system load):

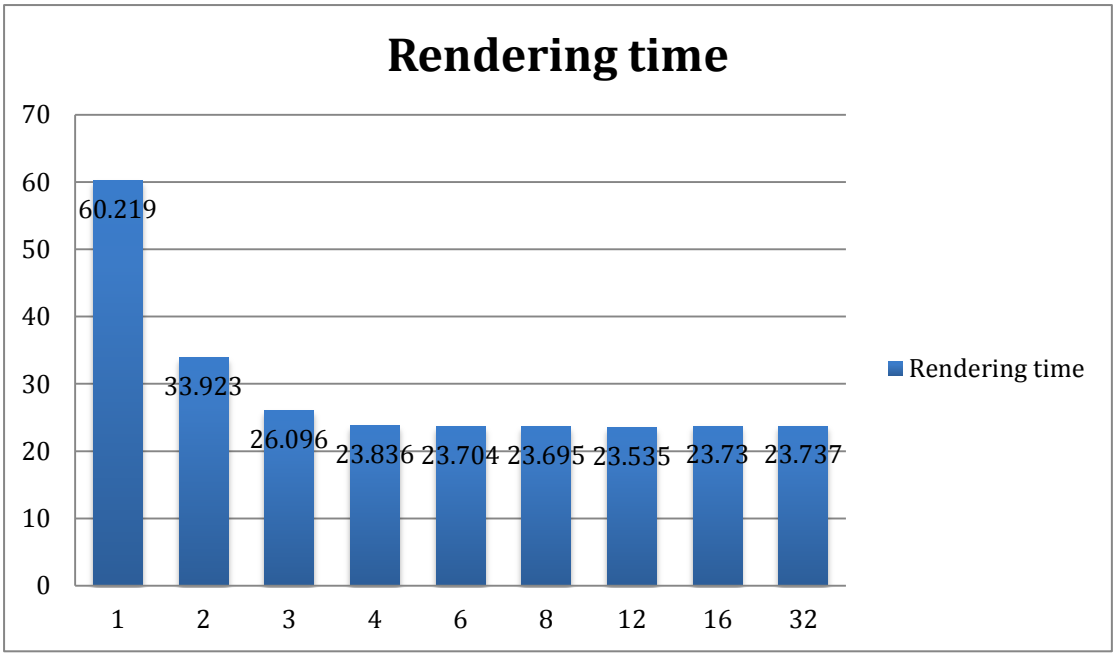

Figure 58 Multi-thread efficiency

From the diagram above, we can see that system can get maximum efficiency at thread number equals to the number of CPU cores.

One of the time-consuming part of Lyrae FX is distributed ray tracing algorithms, because multiple samples are taken instead of one sample.

Soft shadow efficiency is list below (Scene: flower, 249k triangles; 4 threads):

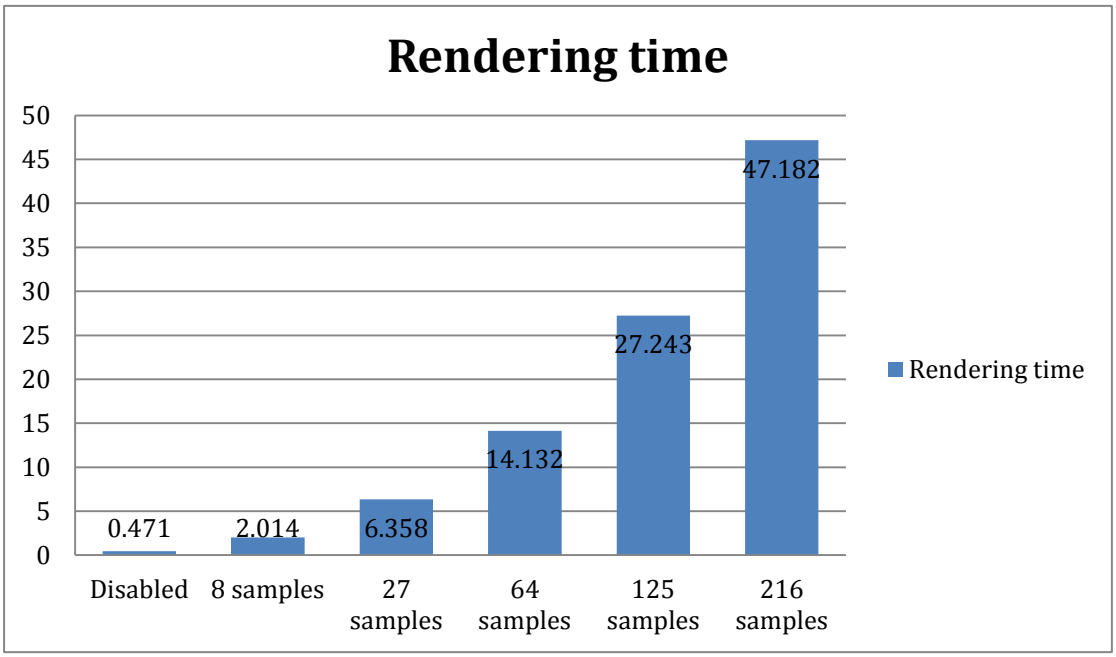

Figure 59 Soft shadow efficiency

As more samples are taken, the rendering time increase dramatically. Usually taking 125 samples is the minimum number to get a smooth soft shadow.

Depth of Fields efficiency is shown below (Scene: spheres):

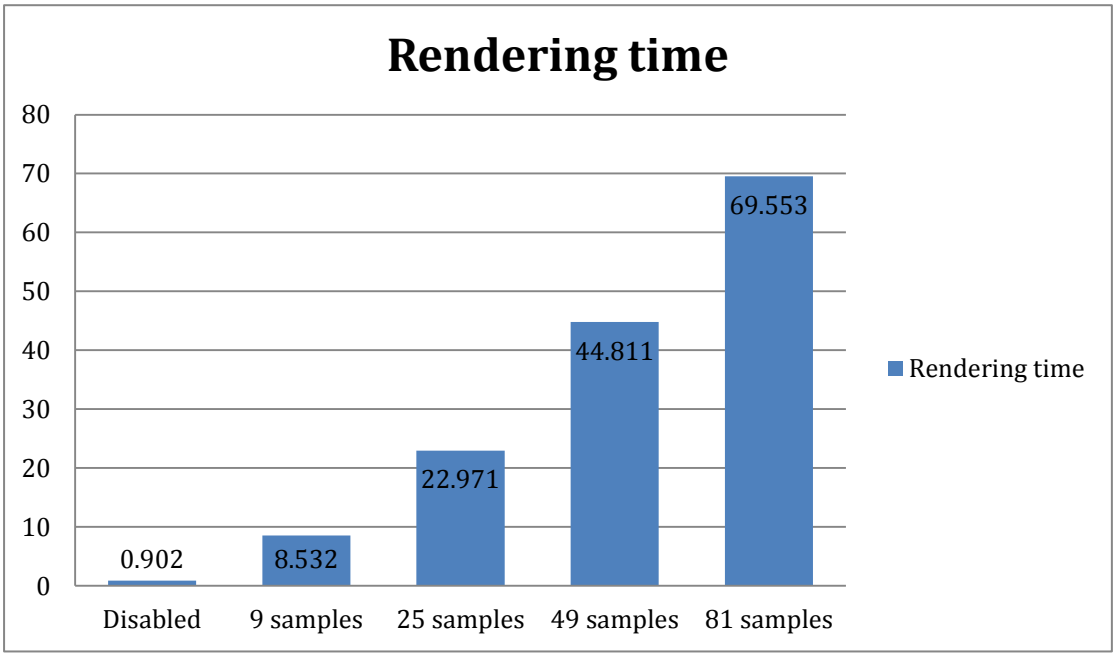

Figure 60 Depth of Field efficiency

Glossy reflection efficiency is shown below:

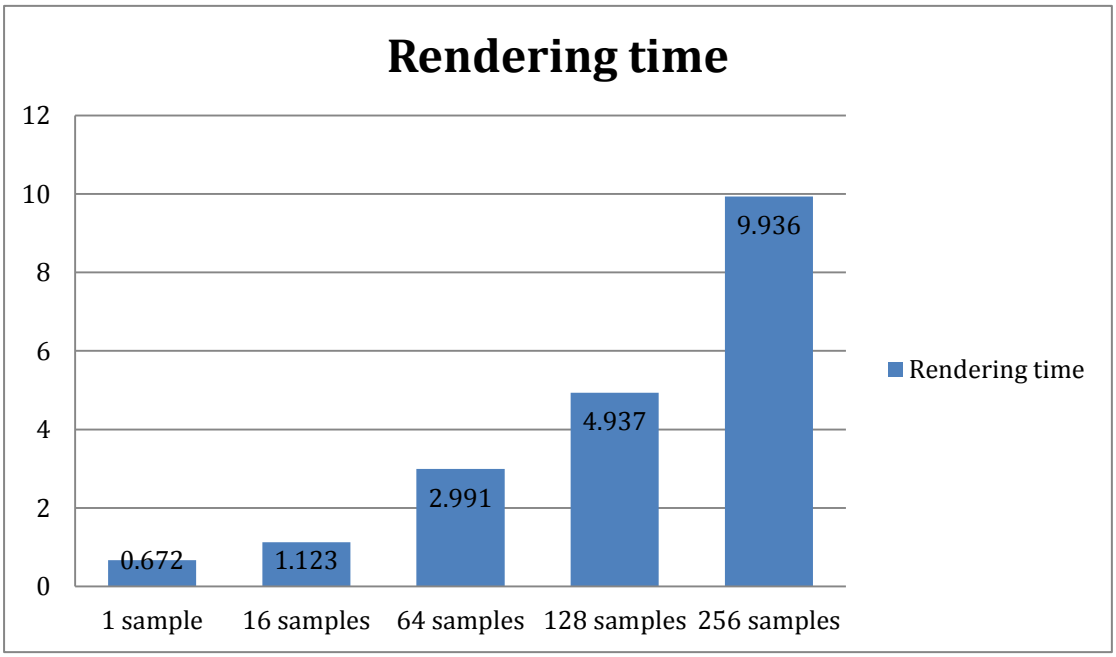

Figure 61 Glossy reflection efficiency

Glossy transmission is similar to glossy reflection, so it's omitted here.

Another part that consumes much more time than ordinary rendering is global illumination. Ambient Occlusion and Photon Mapping all take lots of rendering time. Their efficiency is list below.

Ambient Occlusion efficiency is shown in the following diagram:

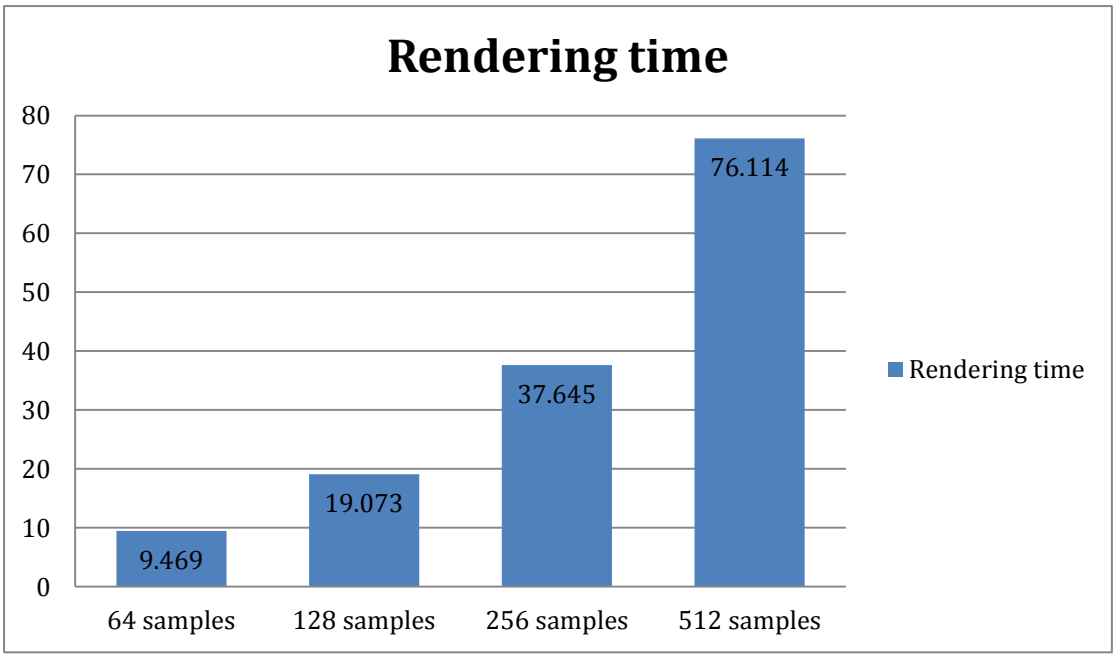

Figure 62 Ambient Occlusion efficiency

Indirect photon mapping also consumes lots of time, because its selection range is quite large. Caustics photon mapping, on the other hand, consumes less time than indirect because its selection range is quite limited, although caustics map quantity is usually much larger than indirect map.

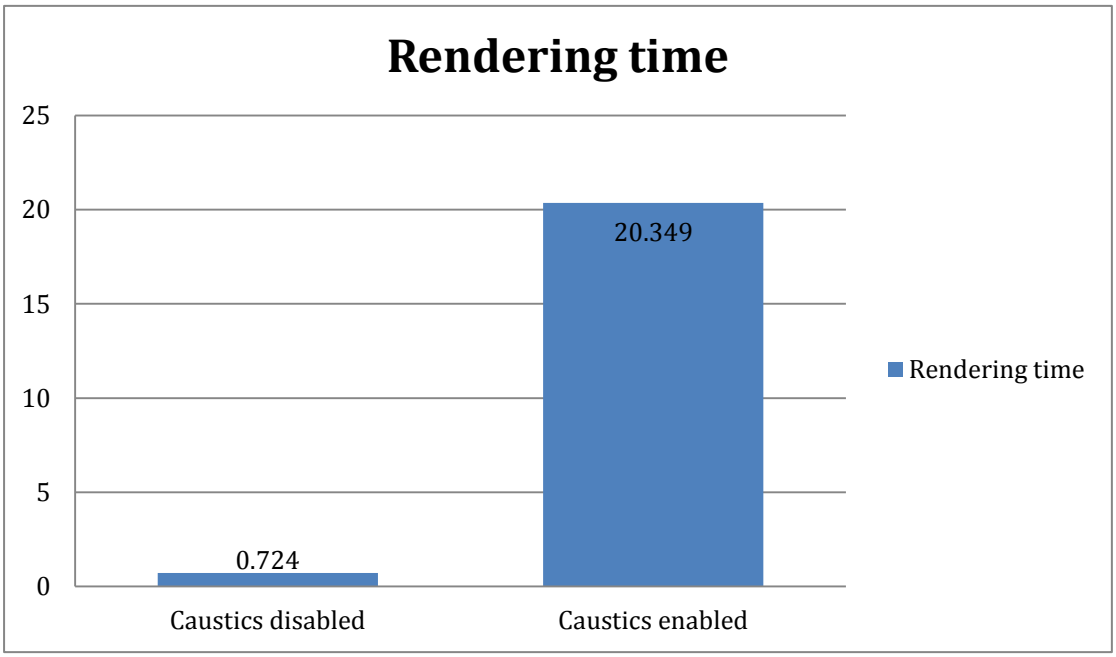

Figure 63 Caustics photon mapping efficiency (1M photons)

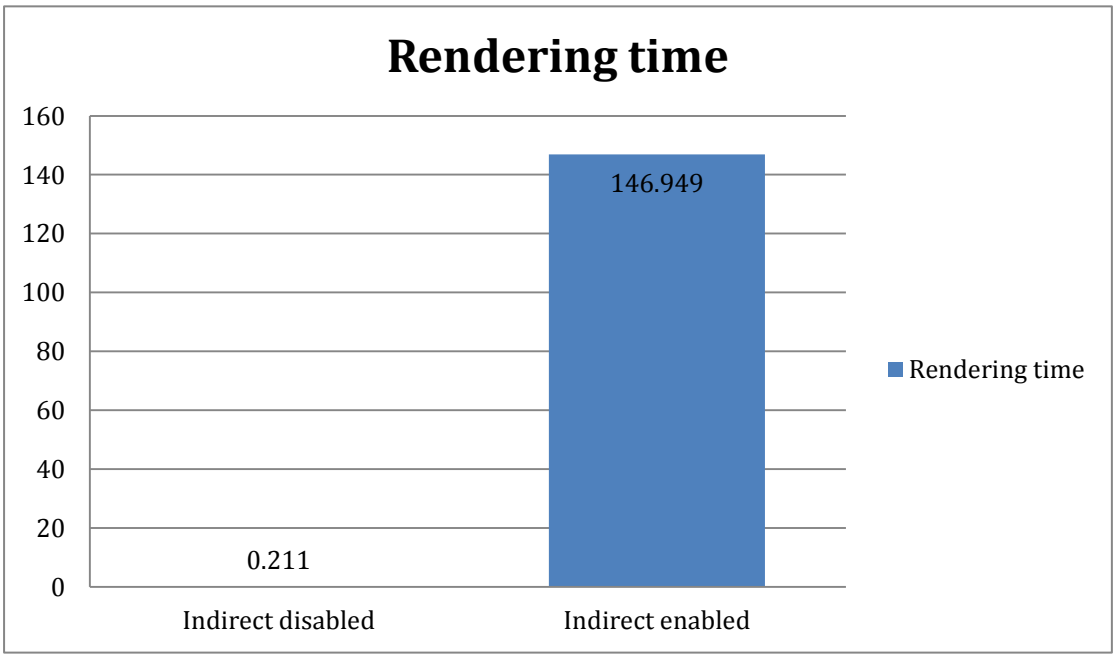

Figure 64 Indirect photon mapping efficiency (50k photons)

Other efficiency comparisons are omitted here, for they are not as important as the ones above.

15. References

[1] Matt Pharr, Greg Humphreys, *Physically Based Rendering*, 2nd ed., 2010
[2] Jacob Bikker, *Raytracing: Theory & Implementation*, 2005,
http://www.devmaster.net/articles/raytracing_series/part1.php
[3] Allen Martin, *Distributed Ray Tracing,*
http://web.cs.wpi.edu/~matt/courses/cs563/talks/dist_ray/dist.html
[4] Zack Waters, *Photon Mapping*,
http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

Appendix A: 3rd Party Libraries

- lib3ds, http://code.google.com/p/lib3ds/
  Lyrae FX uses lib3ds to load 3DS format triangle meshes

- TinyXML, http://www.grinninglizard.com/tinyxml/
  Lyrae FX uses TinyXML to parse XML files and load scene description file

- DevIL, http://openil.sourceforge.net/
  Lyrae FX uses DevIL to load image files

- kd-tree, http://code.google.com/p/kdtree/
  Lyrae FX uses kd-tree in photon mapping

Appendix B: Scene Description File Format

A sample scene description file is listed below:

```
<?xml version="1.0" encoding="utf-8"?>
<LyraeDesc>
    <Config>
        <Float Name="EyePosX" Value="-7" />
        <Vector Name="CameraPosition">
            <Value X="-2" Y="-2" Z="1" />
        </Vector>
        <Bool Name="EnableAdaptiveSampling" Value="1" />
        <Bool Name="EnableSSAA" Value="0" />
        <Int32 Name="SSAASamples" Value="3" />
        <Bool Name="EnableSoftShadow" Value="0" />
        <Int32 Name="SoftShadowSamples" Value="4" />
        <Int32 Name="TargetWidth" Value="800" />
        <Int32 Name="TargetHeight" Value="450" />
        <Bool Name="EnableShadow" Value="0" />

        <Int32 Name="RenderThreads" Value="3" />
        <Float Name="ProjPlaneHalfWidth" Value="4" />
        <Float Name="ProjPlaneHalfHeight" Value="2.25" />
        <Bool Name="DOPEnabled" Value="1" />
        <Float Name="DOPFocalDistance" Value="-2" />
        <Float Name="DOPRange" Value="1.5" />
        <Int32 Name="DOPSamples" Value="3" />

        <Bool Name="EnableSmoothTriangleMesh" Value="1" />
        <Bool Name="EnablePostProcessing" Value="0" />
        <Bool Name="EnablePhotonMapping" Value="0" />
        <Bool Name="EnablePhotonIndirect" Value="1" />
        <Bool Name="EnablePhotonCaustics" Value="1" />
        <Bool Name="PhotonMappingDebugView" Value="0" />
        <Float Name="PhotonCausticsRadius" Value="0.1" />
        <Int32 Name="PhotonIndirectMaximumDepth" Value="4" />
        <Float Name="PhotonAbsorptionProbability" Value="0.2" />
        <Int32 Name="PhotonsIndirectPerLight" Value="10000" />
        <Float Name="PhotonIndirectEnhancer" Value="500" />
        <Float Name="PhotonCausticsEnhancer" Value="2000" />
        <Float Name="PhotonIndirectRadius" Value="5" />
```

```
    <Bool Name="EnableDirectedPhotons" Value="1" />
    <Vector Name="PhotonDirectedP1" >
        <Value X="-40" Y="-40" Z="-40" />
    </Vector>
    <Vector Name="PhotonDirectedP2" >
        <Value X="40" Y="40" Z="40" />
    </Vector>

    <Bool Name="DOPEnabled" Value="0" />
    <Float Name="DOPFocalDistance" Value="-2" />
    <Float Name="DOPRange" Value="1.5" />
    <Int32 Name="DOPSamples" Value="4" />

    <Bool Name="EnableVolumetricLighting" Value="0" />
    <Float Name="VolumetricLightingEnhancer" Value="0.03" />
    <Float Name="VolumetricLightingSampleStep" Value="0.1" />
    <Float Name="VolumetricLightingMaximumDistance" Value="100" />
    <Vector Name="VolumetricLightingP1">
        <Value X="-100" Y="-100" Z="-100" />
    </Vector>
    <Vector Name="VolumetricLightingP2">
        <Value X="100" Y="100" Z="100" />
    </Vector>
</Config>
<Scene>
    <Textures>
        <Texture Name="wood" File="../Textures/wood.jpg" />
        <Texture Name="bg" File="../Textures/hdri20.hdr"/>
    </Textures>
    <Materials>
        <Material Name="bg" Texture="bg">
            <Color R="1" G="1" B="1" />
            <BSDF>
                <Phong Diffuse="0.4" Specular="0" Ambient="0.6">
                    <DiffuseColor R="1" G="1" B="1" />
                </Phong>
            </BSDF>
        </Material>
        <Material Name="simple" Texture="wood">
            <Color R="1" G="1" B="1" />
            <BSDF>
                <Phong Diffuse="0.4" Specular="0" Ambient="0.6">
                    <DiffuseColor R="1" G="1" B="1" />
```

```
            </Phong>
        </BSDF>
    </Material>
    <Material Name="glass" Diffuse="0">
        <BSDF>
            <SpecularReflection>
                <Color R="0.6" G="0.8" B="1" />
                <Fresnel Type="Dielectric" EtaI="1" EtaT="1.5" />
            </SpecularReflection>
            <SpecularTransmission EtaI="1" EtaT="1.5">
                <Color R="0.6" G="0.8" B="1" />
            </SpecularTransmission>
        </BSDF>
    </Material>
    <Material Name="metal" Diffuse="0">
        <BSDF>
            <SpecularReflection>
                <Color R="1" G="1" B="1" />
                <Fresnel Type="Conductor" Eta="1.5" K="3.2" />
            </SpecularReflection>
        </BSDF>
    </Material>
</Materials>

<Shapes>
    <Sphere Material="bg" Radius="20">
        <Position X="0" Y="0" Z="0" />
    </Sphere>
    <Box Material="simple">
        <Position X="-10" Y="-10" Z="0"></Position>
        <Size X="20" Y="20" Z="0.01" />
    </Box>
    <TriangleMesh Material="glass" File="../Meshes/chess.3DS" Scale="0.55">
    </TriangleMesh>
</Shapes>

<Lights>
    <AreaLight Intensity="2">
        <Position X="0" Y="8" Z="3" />
        <Size X="2" Y="2" Z="1" />
        <Color R="1" G="1" B="1" />
    </AreaLight>
</Lights>
```

```
    </Scene>
</LyraeDesc>
```