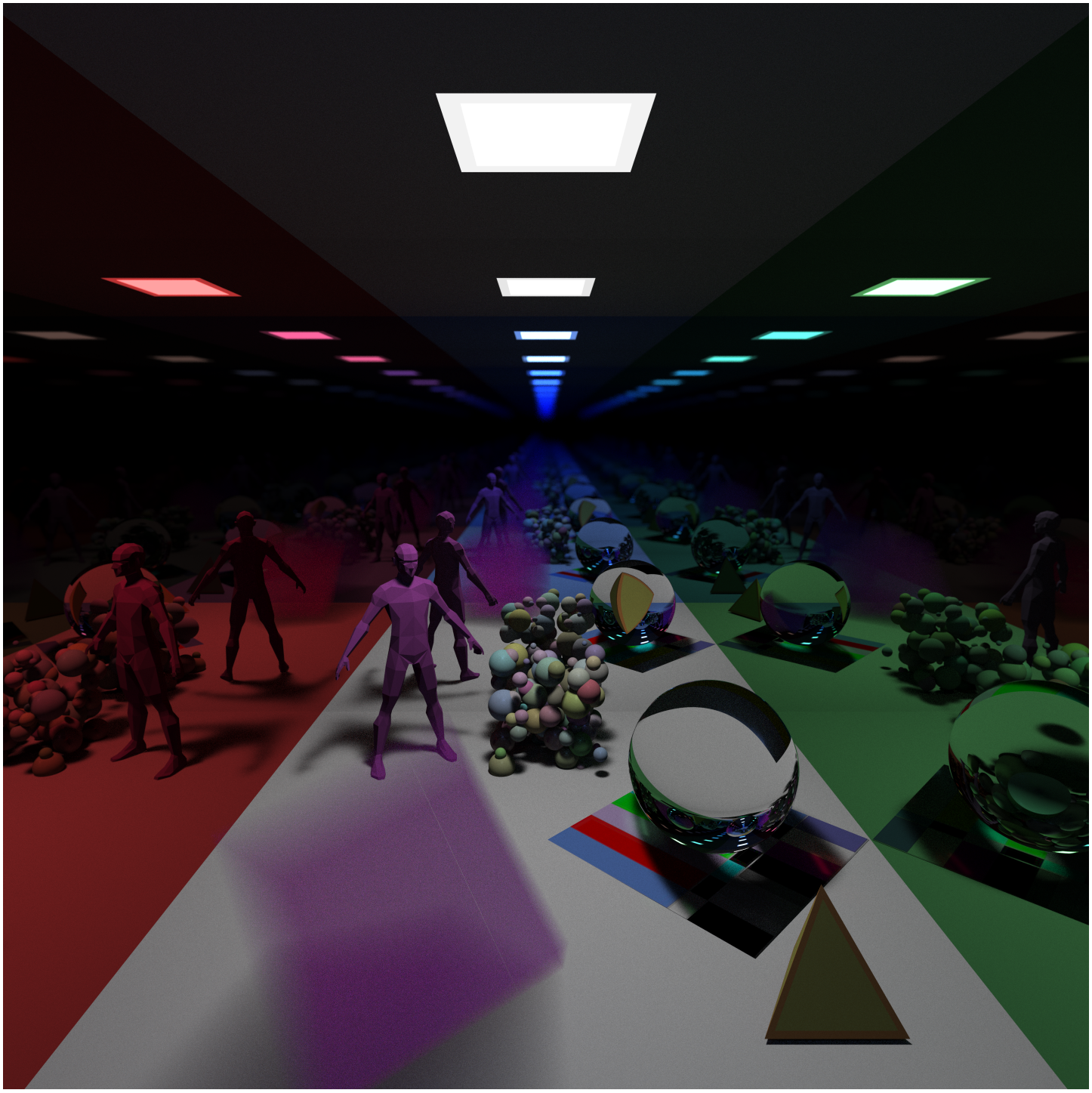
COM S 366: Ray Tracer Final Report

Ryan Leska

December 13, 2024



Contents

1 General Information **2**
- 1.1 CUDA . . . 2
  - 1.1.1 CUDA Call Structure . . . 2
- 1.2 Sampling . . . 2
  - 1.2.1 Implementation - Importance Sampling 2
  - 1.2.2 Testing . . . 2
- 1.3 Octree - Acceleration Datastructure . . . 3
- 1.4 Camera Configuration . . . 3
  - 1.4.1 Depth of Field . . . 3
- 1.5 Anti-Aliasing - Adaptive Sampling . . . 3

2 Materials **3**
- 2.1 Lambertian . . . 3
- 2.2 Metal . . . 4
- 2.3 Dielectric . . . 4
- 2.4 Textured . . . 4
- 2.5 Light . . . 4
- 2.6 Isotropic . . . 4
- 2.7 Other . . . 4
  - 2.7.1 Phong . . . 4
  - 2.7.2 Phong-Lambertian . . . 4
  - 2.7.3 LambertianBordered . . . 5

3 Hittables **5**
- 3.1 Sphere . . . 5
- 3.2 Polygons . . . 5
  - 3.2.1 Triangle . . . 5
  - 3.2.2 Rectangle . . . 5
- 3.3 Boxes . . . 5
- 3.4 Continuous Medium . . . 5
- 3.5 Object Instancing . . . 5
  - 3.5.1 Translation . . . 6
  - 3.5.2 Rotation . . . 6
  - 3.5.3 Motion Blur . . . 6

4 Other Features **6**
- 4.1 Round Pixels . . . 6

5 Feature List Summary - For Easier Grading **6**

1 General Information

This section is very wordy, I understand if you would like to skip it.

generally its a desription of my usage of CUDA, parallelization, Importance Sampling, and Adaptive Sampling. As well as Information about my acceleration datastructure, and camera configuration.

1.1 CUDA

For my Ray Tracer (Path Tracer) Implementation, I decided to use CUDA c++ to parallelize the rendering process. This lead to a lot of headaches, dealing with memory issues and host-device scope issues, but in the end it seems to have sped up the rendering process significantly.

The majority of my code runs on the device (gpu), but the host (cpu) is used to read in files like textures and polygon meshes. The host is also used to write the final image to a file.

During testing I was able to run my program on a Nvidia 4070 ti and a Nvidia 3070 mobile. From this testing I determined that the best configuration for both of the gpus was to have 128 threads allocated per block (block count is calculated based on the number of pixels in the image). I generally chose to have my threads in a 16x8 because I believe that this increases the likelihood of cache hits.

1.1.1 CUDA Call Structure

First the main function allocates random number generators on the gpu (one for each thread). Then it reads in textures and polygon meshes on the host (cpu), then the data is allocated and copied to the gpu memory. The main function then allocates memory for the world and camera on the gpu. The world is then populated with hittables and materials. Then the render is ran on the gpu. After the render is complete the image data is copied from the gpu to the host and written to a file.

1.2 Sampling

1.2.1 Implementation - Importance Sampling

The sampling method I settled on for my final implementation was a weight cosine hemisphere sample along side a weighted direct light sample. This method was chosen because it seemed to produce the best results in terms of noise and speed.

When a scattering material is hit the ray has an equal (or configurable) chance to either scatter in a cosine weighted direction or towards a light. The light is chosen randomly from the list of lights in the world. Both of these samples are then weighted by their respective pdfs are applied to the final color.

Due to my usage of CUDA my path tracing function getColor() is not recursive. Instead I used an iterative approach where the ray is scattered until it hits a light or the maximum depth is reached.

1.2.2 Testing

During my testing I tried a bunch of different sampling methods. I tried a square random sample, a uniform sample, a cosine weighted sample, a power-weighted cosine sample, a Beckmann distribution sample, and a Blinn-Phong sample. I found that the a cosine weighted sample produced the best results in terms of lighting. Below is a test image I produced using each of these sampling methods.

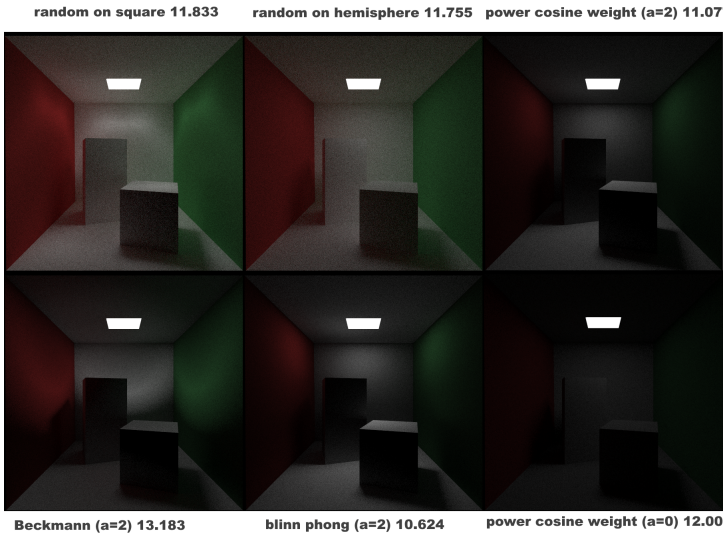


Figure 1: Sampling Test

1.3 Octree - Acceleration Datastructure

(/lib/hittable/Octree.cu (.hpp))

To help with the speed of my ray tracer I implemented an octree to store the hittables in the world. This octree is built when the world is created. Since I am using CUDA C++ I made the Octree a child of the scene (world) class. The octree is built by recursively splitting the world into 8 octants until the number of hittables in the octant is less than a threshold or a configurable depth is reached.

While working on the octree I though about how a voxel based render might be implemented to be really fast, so I might try that in the future.

1.4 Camera Configuration

(/lib/processing/Camera.cu (.hpp))

The camera is configured by setting a number of parameters during the world/camera creation. The camera is configured by setting the following parameters:

1. fov - the field of view of the camera
2. lookfrom - the position of the camera
3. lookat - the position the camera is looking at
4. vup - the up vector of the camera (used to determine the orientation of the camera)
5. aspect ratio - the aspect ratio of the image (This can override the resolution)
6. apature - the size of the apature (used to determine the depth of field)
7. focus distance - the distance the camera is focused at (used to determine the depth of field)
8. MSAA - The scalar of samples per pixel when the camera detects an edge or aliased section
9. samples - The number of samples per pixel (pre-msaa)
10. bounces - the number of bounces the ray will take before it is terminated
11. Ambient Light - the percent brightness of the ambient light in the scene

1.4.1 Depth of Field

The depth of field is implemented by using the apature and focus distance. When rays are being cast they are randomly offset by the apature size then the focus distance is used to determine the points the rays are focused on.

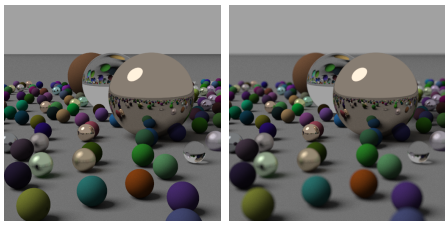


Figure 2: Left: No Depth of Field, Right: Depth of Field

1.5 Anti-Aliasing - Adaptive Sampling

For Adaptive Sampling I implemented a method similar to MSAA that detects edge and aliased sections (like textures) and increases the number of samples taken in that area. This method is implemented by a simple edge detection algorithm in the hit function. If the hit function detects an edge or aliased section it will increase the number of samples taken in that area.

2 Materials

(/lib/Materials/Materials.cu/.hpp)

In my design Materials are child classes of the Material class. This call has a couple different methods that are used in the rendering but the primary method is the scatter method which determines how to color and redirect the ray.

2.1 Lambertian

(Lambertian.cu/.hpp)

A lambertian material is a material that scatters light in all directions following Lambert's cosine law. This material is used to simulate a matte surface.

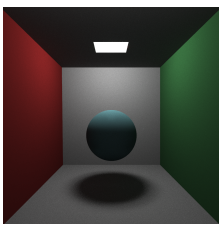


Figure 3: Lambertian Sphere

2.2 Metal

(Metal.cu/.hpp)

A metal material is a material that reflects light in a specular direction. This material is used to simulate a shiny surface (fuzzy or clear).

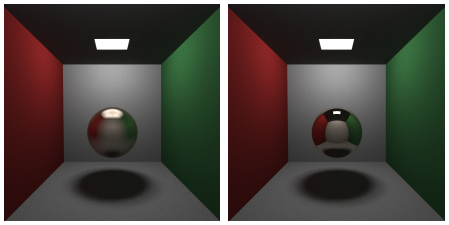


Figure 4: Left: Fuzzy Metal Sphere, Right: Clear Metal Sphere

2.3 Dielectric

(Dielectric.cu/.hpp)

A dielectric material is a material that refracts light when it hits the surface. This material is used to simulate glass or water.

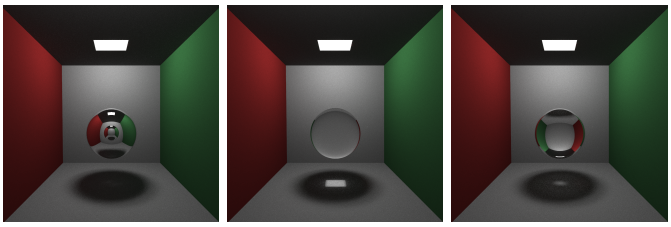


Figure 5: Left: k = 0.3, Middle: k = 1.5, Right: k = 5

2.4 Textured

(Textured.cu/.hpp)

A textured material is a material that has a texture applied to it. This material is used to simulate surfaces with patterns or images. This material uses u,v mapping to determine the texture location. PPM textures are supported and must be read in by the cpu.

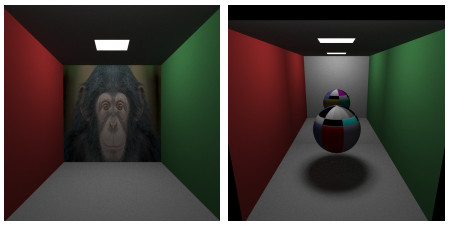


Figure 6: Left: Textured Monky Polygon, Right: Textured TV-TEST Sphere

2.5 Light

(Light.cu/.hpp)

A light material is a material that emits light. This material is used to simulate light sources in the scene.

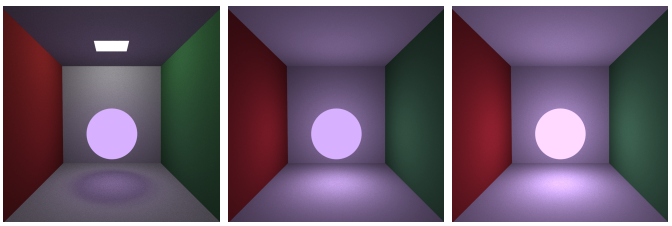


Figure 7: Left: Lit Light Sphere, Middle: Unlit Light Sphere, Right: Bright Unlit Light Sphere

2.6 Isotropic

(Isotropic.cu/.hpp)

An isotropic material is a material that scatters light in all directions. This material is used to simulate a matte surface. The way this differs from lambertian is that Isotropic sattering is in a sphere while lambertian is on the hemisphere

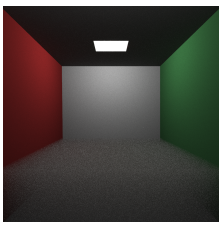


Figure 8: Continuous Medium (Smoke)

2.7 Other

2.7.1 Phong

(Phong.cu/.hpp)

This material was lightly discussed in the class. I implemented it as a test, but it is not used in the final render. The material is a shaded material following the Phong Lighting model.

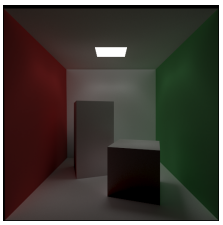


Figure 9: Phong Lighting Cornell Box

2.7.2 Phong-Lambertian

(PhongLamb.cu/.hpp)

This material is a combination of a Phong and Lambertian material. The material is passed a parameter that determines a number of bounces before the material switches to a Phong material. This allows for the nice uniform phong lighting along with color bleeding and light reflections from nearby objects. This results in a similar result to Importance sampling with light sampling.

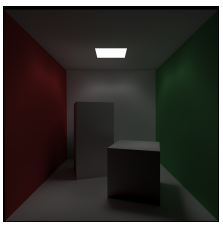


Figure 10: Phong-Lambertian Cornell Box 2 Bounce Phong Limit

2.7.3 LambertianBordered

(LambertianBordered.cu/.hpp)

A lambertian material that returns a different color when an edge is hit. I created this during my testing of MSAA. I found the effect cool, its very similar to the cell shading seen in the video-game series Borderlands.

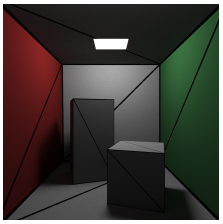


Figure 11: Lambertian Bordered Cornell Box

Note: these boxes are polygons not the boxes hittable object. For a better cell shading effect.

3 Hittables

(/lib/hittable/ (/headers/hitable.hpp))

Hittable is misspelled in the code, I apologize for that.

Hittables are objects that can be hit by a ray. Continuing the trend of OOP in my code I made hittables a parent class of all hittable objects. Each hittable object has a hit function that determines if a ray hits the object and returns a hit record (see HitRecord.hpp in hittable/headers/).

3.1 Sphere

(Sphere.cu/.hpp)

A sphere is a hittable object that is defined by a center and a radius.

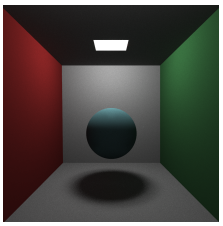


Figure 12: Lambertian Sphere

3.2 Polygons

(Polygon_T.cu/.hpp)

A polygon is a hittable object that is defined by a list of vertices in a plane. I implemented this such that the polygons can be any number of vertices and these vertices can be texture mapped using u,v coordinates.

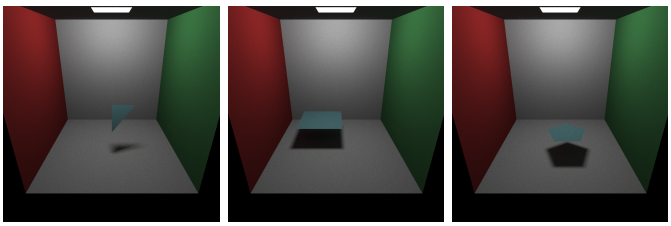


Figure 13: Left: Triangle, Middle: Rectangle, Right: Pentagon

3.2.1 Triangle

To ease my scene creation I implemented a method Triangle that is passed 3 vertices and a material. This method creates a triangle from the vertices and material. (see first graphic above)

3.2.2 Rectangle

To ease my scene creation I implemented a method Rectangle that is passed 4 vertices and a material. This method creates a rectangle from the vertices and material. (see second graphic above)

3.3 Boxes

(Box.cu/.hpp)

A box is a hittable object that is defined by a minimum and maximum point.

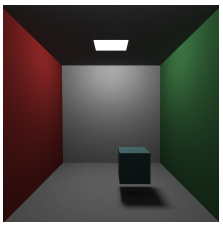


Figure 14: Box

3.4 Continuous Medium

(Medium.cu/.hpp)

A continuous medium is a hittable object that is defined by a boundary and a density. This object can be used to simulate fog or smoke (using the isotropic material).

(see Isotropic material above for image)

3.5 Object Instancing

(ObjInst.cu/.hpp)

Object Instancing are classes that are hittable in themsleves but contain other hittables. This allows for the creation of more complex objects from simpler objects.

For each of the following classes I used ray inverse transformations to determine if a ray hit the object.

3.5.1 Translation

Translation is a class that is a hittable object that contains another hittable object.

3.5.2 Rotation

Rotation is a class that is a hittable object that contains another hittable object.

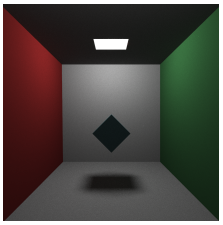


Figure 15: Box translated and rotated

3.5.3 Motion Blur

Motion Blur is a class that is a hittable object that contains another hittable object. It is passed a Velocity vector, an Acceleration vector, and a time interval. The time interval determines both how far the object moves as well as the number of overall hit attempts per ray hit check. A long time leads to more sample and therefore a clearer movement (with respect to a lower time higher velocity).

Bounding boxes for motion blur are done by sampling the object throughout its movement and creating a bounding box from the maximum and minimum points.

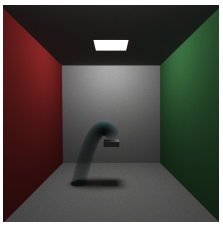


Figure 16: Ball up left velocity, down acceleration

4 Other Features

4.1 Round Pixels

To make the image look better I implemented round pixels. When compiling the user can set a flag to either use a circular montecarlo sample (round pixel) or a square montecarlo sample (square pixel).

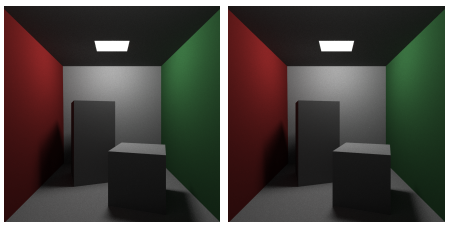


Figure 17: Left: Square Pixels, Right: Round Pixels

5 Feature List Summary - For Easier Grading

Required

◇ Camera Configuration

Position, Orientation, Field of View

◇ Anti-Aliasing

configurable sample count

See also Adaptive Sampling

◇ Ray/Sphere Intersection

◇ Ray/Triangle Intersection (Polygon)

◇ Load texture (PPM format)

◇ Textured Spheres and Triangle (both have u,v mapping)

◇ Acceleration Datastructure (Octree)

◇ Specular (Metal), Diffuse (Lambertian), and Dielectric Materials

◇ Emissive Material (Light)

For Points

◇ Volume Rendering (smoke, clouds)

◇ Quads (Rectangle, Box)

◇ Motion Blur

◇ Defocus/Depth of Field (see camera configuration)

◇ Object Instancing (Translation, Rotation)

◇ Importance Sampling (Cosine weighted and Light Sampling Mix)

◇ Round Pixels

◇ Parallelization (CUDA)

◇ GPU Acceleration (CUDA)

◇ Adaptive Sampling (MSAA)

For Fun

◇ Phong lighting model

◇ Phong-Lambertian Material

◇ LambertianBordered Material

◇ Other Sampling Methods (Beckmann, Blinn-Phong, Square, Uniform)