# CPSC 490 Spring 2017:
# Extended Path Tracing Exploration

Wendy Chen
Advisor: Professor Holly Rushmeier

May 4, 2017

**Abstract**

In the fall semester, my path tracing studies were focused on understanding the theory behind Kajiya's Rendering Equation and how path tracing solves the integral. My study in particular focused on various sampling techniques that could be used for Monte Carlo Integration. From my studies, I built a unidirectional path tracer. This semester, I have focused on implementation, system design, and extending the features of the path tracer.

## 1 Introduction

While the focus of last semester was on path tracing theory and sampling techniques used in Monte Carlo Integration, the focus of this semester has been on extending the path tracer system developed from the last semester. Through the work I have done on implementation this semester, I have focused on system design and infrastructure. The rest of the report will discuss various aspects of the implementation. Several extended features of the path tracer are examples of distributed ray tracing. These features, such as sampling area lights or simulating depth-of-field, will be discussed in further detail later in the report.

## 2 Compilation and Rendering

For a clearer understanding of the implementation details that will be discussed, the reader may wish to view and compile the source code. The source code can be viewed at https://github.com/vverovvero/cs490sp/tree/master.

The path tracer was developed on MacOSX and is written in C++. The png image conversion depends on the Cairo image library. To run the source code on your own machine, you may need to install Cairo and change its directory link in the Makefile.

After compiling, the program can be run with the command:

```
./main -r [low | high] [input bin file] [output png file]
```

The resolution flag allows a user to select low or high sampling, where low sampling takes one sample per pixel and high sampling takes twenty-five stratified samples per pixel. The input is a binary (.bin) file, which will be generated from a scene (.sce) file. The output is a Portable Network Graphics (.png) file.

The user editable file is the scene file, which is then used to generate the binary file that is read by the program.

# 3 Designing Scene Descriptions for Users

In order to render more interesting scenes, the path tracer needed to be extended so that complex geometry could be supported. To allow the user to render arbitrary geometry, I designed a scene description file to be used by the path tracer system, and I built a scene parser.

The goals behind the scene description design were as follows:

- The scene description should not be dependent on existing modeling software.
- The scene description should be easily editable, and thus support a restricted set of primitives.
- The parsing of the scene description should be removed from the C++ program.

The path tracer was originally implemented without the idea that it might depend on pre-existing modeling software. To keep the path tracer free of depending on scene input that might be susceptible to external software changes, I decided to design a minimal scene specification language and file extension.

While there is no GUI to preview how the scene looks while a user builds it, I wanted to make the scene description as easily editable as possible. The scene file is simply a text file, where each line represents a distinct primitive in the scene. Each line begins with a keyword declaring a primitive, followed by the primitives attributes. The user can easily add or remove primitives line by line. Because all scene manipulation must be done through text, the primitives available to the user are restricted. The user is allowed to specify the film size, place the camera, instantiate lights, create materials, add triangles or spheres to the scene, and give the scene a bounding box. The user may declare primitives in any order they wish, except that materials must be declared in order and before any geometry.

Because the user is not restricted in the ordering of primitives in their scene description, parsing the text file becomes more complicated. Ideally, the path tracer should expect a file format of ordered information. The path tracer should not have the responsibility of parsing user text or sanitizing input. An intermediate parser is introduced to create a binary file from the user's scene description. The path tracer expects information from the binary file in an ordered manner and can simply initialize all primitives while reading in the binary file. Thus, the user's scene description is one layer removed from the actual path tracer.

## 3.1 Scene Description Definition

The primitives supported in the scene description file are: `FILM, CAMERA, LIGHT, MATERIAL, SPHERE, TRIANGLE, INCLUDE, BOUNDINGBOX`.

A valid scene description must include `FILM`, `CAMERA`, and `BOUNDINGBOX` primitives. Although the parser will not complain if the other primitives are missing, the user will not see an image if no lights, materials, or geometry are placed in the scene.

Each line in the text file represents a primitive in the scene. Each line must begin with a primitive keyword. The attributes for each primitive are space-separated after the keyword. The arguments that each attribute takes are comma-separated within parentheses after the attribute.

The `INCLUDE` primitive allows a user to include complex meshes that would be difficult to add by hand but can be generated using the one of the intermediary scene parsers. Because `INCLUDE` primitives are

```
FILM Width(512) Height(512)
CAMERA Point(50, 50, 200) LookAt(50, 25, 0) FOV(60) Up(UP)
LIGHT Type(OMNI) Point(50, 95, 50) Color(180, 180, 180)
MATERIAL Color(255, 255, 255) Type(ORIGINAL) Specular(0) Lambert(0.85) Ambient(0.05)
SPHERE Point(70, 25, 50) Radius(25) Material(0)
TRIANGLE P1(0, 0, 0) P2(100, 0, 0) P3(0, 100, 0) Material(0)
BOUNDINGBOX Min(-500, -500, -500) Max(500, 500, 500)
```

Figure 1: Simple scene description (.sce) file

| **Primitive** | **All Possible Attributes** |
|---|---|
| FILM | Width(i) Height(i) |
| CAMERA | Point(f,f,f) LookAt(f,f,f) FOV(f) Up(f,f,f) LensRadius(f) FocalDepth(f) |
| LIGHT | Type(OMNI\|SPOT\|SPHERICAL) Point(f,f,f) Color(f,f,f) ToPoint(f,f,f) Angle(f) FallOffAngle(f) Radius(f) |
| MATERIAL | Color(f,f,f) Type(ORIGINAL\|PHONG) Specular(f) Lambert(f) Ambient(f) Metal(bool) Exponent(f) |
| SPHERE | Point(f,f,f) Radius(f) Matieral(i) Translate(f,f,f), ScaleR(f), RotateX(f), RotateY(f), RotateZ(f) |
| TRIANGLE | P1(f,f,f) P2(f,f,f) P3(f,f,f) Material(i) Translate(f,f,f), Scale(f,f,f), RotateX(f), RotateY(f), RotateZ(f) |
| INCLUDE | Mesh(file location) + TRIANGLE or SPHERE transformation attributes |
| BOUNDINGBOX | Min(f,f,f) Max(f,f,f) |

Figure 2: Table of supported attributes for each primitive (i for integer argument, f for float argument)

supported, users can convert any OBJ file into a list of `TRIANGLE` primitives, which are supported by the restricted scene language.

For further description of the appropriate use of the attributes and their arguments, please see the README in the source code.

## 3.2 Intermediary Representations

The parsing stages are broken into intermediate steps in order to keep parsing direct user input separate from the path tracer. Python is used to implement the intermediate parsers.

To allow a user to `INCLUDE` complicated meshes, the intermediate parsers can convert OBJ (.obj) files into a text (.txt) representation of a vertex array and face array. This text representation can then be converted into a scene description (.sce) file that can be included in the main scene file. Because the conversion from an OBJ file depends on which modeling software was used to generate the OBJ file, the intermediate parsing step requires that the OBJ files be generated by Blender.

The parser for converting OBJ files into intermediate text representations can be called using:

```
python ./OBJparser.py [input obj file] [output txt file]
```

The parser for converting the intermediate text file into a scene (.sce) description file can be called using:

```
python ./TXTparser.py [input txt file] [output sce file]
```

Finally, the parser to convert the user scene description file into a binary file for the path tracer to read

can be called using:

```
python ./SCEparser.py [input sce file] [output bin file]
```

The scene parser reads the user scene description file and builds an internal representation of the scene using Python dictionaries. The internal scene representation is then serialized into binary by sending integer codes that correspond to each primitive, followed by the arguments to their attributes in a set order. Because the scene description file is always serialized in a set manner, the path tracer can always know how many arguments to expect after reading in each primitive integer code.

### 3.3 User Interface

By separating the internal scene representations (in both Python and C++) from the user scene description file, a clean user interface is achieved.

The user interacts with the path tracer in a two-step pass, by first parsing the scene description file into a binary file, and then rendering with the binary file. For ease of running the system, shell scripts have been provided:

```
./makebin.sh [file.sce] [file.bin]

./makepng.sh [file.bin] [file.png] [res=low|high]
```

### 3.4 Alternative Scene Representations

While I chose to design and implement custom scene description files and scene parsing, other solutions for scene representation include specifying scenes in JSON. Pre-existing parsers could then be used to parse the JSON into Python dictionaries. Once the scene is represented internally as Python dictionaries, the scene can be serialized and fed into the path tracer as described in previous sections.

Using JSON and pre-existing parsers would have saved the trouble of defining a custom user scene description file and parsing the file into Python, but it does not solve the question of how a user should then be able to include complex meshes.

## 4 Acceleration Structures

In order to support rendering complex geometry, an acceleration structure is necessary to relieve the system of needing to search through all geometry when checking for intersections with each ray. A kd-tree has been implemented to speed up the process of checking for scene intersections. The kd-tree is implemented as a binary tree, where each node represents a bounding box and the associated pieces of geometry that lie inside of or intersect the box. Each node's bounding box is split down its midpoint along its longest axis to form its children nodes and bounding boxes. The depth of the tree is capped.

The kd-tree is constructed immediately after the path tracer reads in the input binary file and creates an internal scene representation. `KDnode` and `KDtree`, the classes that implement the kd-tree, can be found in `SCEparser.cpp`. The code used for testing triangle-box intersections was adapted from code written by Tomas Akenine-Moller.[1]

As mentioned, naive splitting down the midpoint of the bounding box is used for creating the children nodes. Naive splitting works best when geometry is well distributed through the scene. If geometry is

clumped together, a better splitting method would split along the average midpoint of the objects rather than the midpoint of the bounding box itself. However, the benefit of naive splitting is that the bounding boxes will always subdivide–there is no risk that the midpoint will be picked such that a child node is as large as its parent.

### 4.1 Alternative Splitting Methods

An alternative splitting method would be to split along the average midpoint of the objects. Besides the need to define what the midpoint of a triangle should be considered, this splitting method runs the risk of not being able to substantially shrink the size of the child node. When this splitting method was tried, it was found to be ineffective due to the fact that some triangles that intersected with the bounding box had midpoints that were outside of the bounding box itself. Because outlying midpoints were taken into consideration when finding the average midpoint of a node, the average midpoint would often cease to change, and the same point would be selected repeatedly.

Other splitting methods include using the surface area heuristic, which requires comparing ratios of surface areas rather than looking at object midpoints.[2]

## 5 Threading

In addition to implementing acceleration structures in order to support complex geometry, threading was also implemented to parallelize the main render loop. The computation behind tracing each path is independent from one another, so the computations for rendering each pixel can be run at the same time without concurrency issues.

The implemented threading approach is to divide the image into horizontal blocks among the threads. Thus, the image film height must be divisible by the number of threads. Each thread renders into its own image buffer, and the buffers are stitched together into a single image afterward.

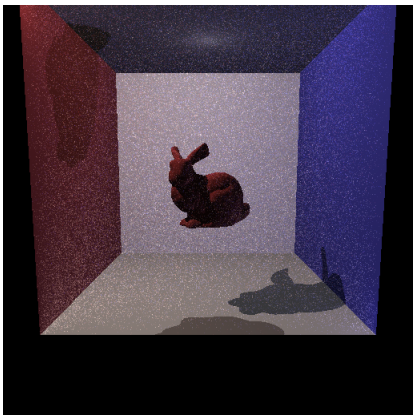
The C++ std::thread library was used. Because std::thread requires a constant number of threads, the program must be recompiled if the user desires to change the number of threads spawned. The default number of threads the path tracer uses is eight threads. The Mac used for development has a 2.5 GHz Intel Core i7 Processor, which has four processor cores and can optimally support eight threads.

For a sample scene with a floating bunny, the timing data shows that increasing the number of threads beyond eight leads to increased performance, but with very quickly diminishing returns.

## 6 Monte Carlo Integration

In path tracing, we must take integrals over areas that cannot be computed analytically. To solve such integrals, we can take samples and estimate the value of the integral using Monte Carlo Integration. As explained in more detail in my report from last semester, the Monte Carlo estimator converges to the the true value of the integral. In this section, we discuss various areas of implementation that rely on being able to pick uniform samples in order to distribute rays.

From last semester, Monte Carlo Integration allowed paths to be generated in the unidirectional path tracer. To generate paths, we must pick a new direction to send the ray towards after each bounce. The new direction was picked by sampling uniformly from a hemisphere. For more details on path construction



| Number of Threads | Render time (seconds) |
|---|---|
| 1 | 266.440656 |
| 2 | 190.780939 |
| 4 | 135.698511 |
| 8 | 116.786129 |
| 16 | 105.386543 |
| 32 | 98.289859 |
| 64 | 96.909107 |

Figure 3: 'Floating bunny' scene rendered with varying number of threads, 1 sample per pixel

(and a more general discussion of how path tracing solves Kajiya's Rendering Equation), please refer to the report from last semester.

### 6.1 Sampling Area Lights

The point light is a basic model where all light originates from a single point. When objects are illuminated by a point light, they cast hard shadows. Spotlights are similarly a variation of a point light, except for the addition of a realistic tapering off of light around the rim of the spotlight's cone of directions.

To achieve more realistic soft shadows, we need to implement area lights. In my path tracer, I have implemented spherical area lights. The light struct and user input are updated to include a field for the radius of a spherical light.

To sample from an area light, it must be possible to uniformly pick a sample from the light's total area. In the case of a spherical area light, samples are taken uniformly over a unit sphere. These samples are scaled by the light radius and used as an offset from the light's center point.

An optimization could be made to only sample over the solid angle subtended by the spherical light.[3, p. 720]. In my implementation, I assume the spherical light can be sampled over its entire area regardless of the direction from which it is being sampled.

### 6.2 Sampling Depth of Field

The basic path tracer approximates the camera as a pinhole camera, where all rays that enter the scene originate from a single point. However, assuming that the camera is always a pinhole camera is not realistic.

Real lenses on cameras have a certain amount of area through which we can shoot rays. To get realistic blurring and depth of field effects, we approximate the camera lens as a disk. The camera struct and user input are updated to include parameters for lens aperture size and the desired focusing depth along the z-axis.

To implement depth of field, we first draw a uniform random sample from a unit disk. Scaling this sample by the camera lens radius, we use the sample to offset the ray we will shoot into the scene from the camera point.

Next, we would like to adjust the ray direction such that the ray intersects at the same point on the plane of focus as an unperturbed ray would be. We first compute the intersection point for an unperturbed ray on the plane of focus. We use this intersection point to update the direction we send our perturbed ray. By guaranteeing that our perturbed rays have the same intersection points as unperturbed rays on the plane of focus, we ensure that objects along the plane of focus will remain clear.

We illustrate how to uniformly sample a unit disk, because the intuitive answer is not correct for flat disks.[3, p. 665] Intuitively, a disk can be sampled for in its polar coordinates with $r = \xi_1$ and $\theta = 2\pi\xi_2$. However, taking such samples will be biased towards the center of the disk. Below, we derive how uniform disk samples can be taken.[3, p. 665 - 668]

## Uniform Sampling of a Unit Disk

1. Due to uniform sampling with respect to area, the PDF is constant:

$$\begin{aligned} p(x,y) &= c \\ \int p(x,y)dxdy &= 1 \\ c\pi r^2 &= 1 \\ c &= \frac{1}{\pi} \\ p(x,y) &= \frac{1}{\pi} \end{aligned}$$

2. From transforming between Cartesian and polar coordinates, we know:

$$\begin{aligned} p(r,\theta) &= rp(x,y) \\ p(r,\theta) &= \frac{r}{\pi} \end{aligned}$$

3. Calculating marginal and conditional densities.

$$\begin{aligned} p(r) &= \int_0^{2\pi} p(r,\theta)d\theta \\ &= \int_0^{2\pi} \frac{r}{\pi} d\theta \\ &= \frac{r}{\pi}\theta\Big]_0^{2\pi} \\ &= 2r \\ p(\theta|r) &= \frac{p(r,\theta)}{p(r)} \\ &= \frac{r}{\pi} \cdot \frac{1}{2r} \\ &= \frac{1}{2\pi} \end{aligned}$$

4. Integrate to get CDFs and get inverse.

$$\begin{aligned} P(r) &= \int_0^r p(r')dr' \\ &= \int_0^r 2r'dr' \\ &= r'^2\Big]_0^r \\ &= r^2 \\ P^{-1}(r) &= \sqrt{r} \end{aligned}$$

$$\begin{aligned} P(\theta|r) &= \int_0^\theta p(\theta'|r)d\theta \\ &= \int_0^\theta \frac{1}{2\pi} d\theta \\ &= \frac{\theta}{2\pi} \\ P^{-1}(\theta|r) &= 2\pi\theta \end{aligned}$$

5. Correct sampling is:

$$\begin{aligned} r &= \sqrt{\xi_1} \\ \theta &= 2\pi\xi_2 \end{aligned}$$

# 7 Images

The following is a gallery of images rendered with my unidirectional path tracer. Each image is 512 x 512 pixels, rendered with eight threads with twenty-five samples taken per pixel. The render times for each image are provided in a table as well.

The selected images demonstrate particular features of this path tracer, such as its support for complex geometry, depth of field, and area lights.

| **Image** | **Render time (seconds)** |
|---|---|
| Figure 5. Three Spheres | 1010.536735 |
| Figure 6. Bunny and Teapot | 3577.509953 |
| Figure 7. Spheres, focal depth at 50 | 978.591085 |
| Figure 8. Spheres, focal depth at 140 | 1006.195048 |

Figure 4: Render time in seconds for the high-resolution images.

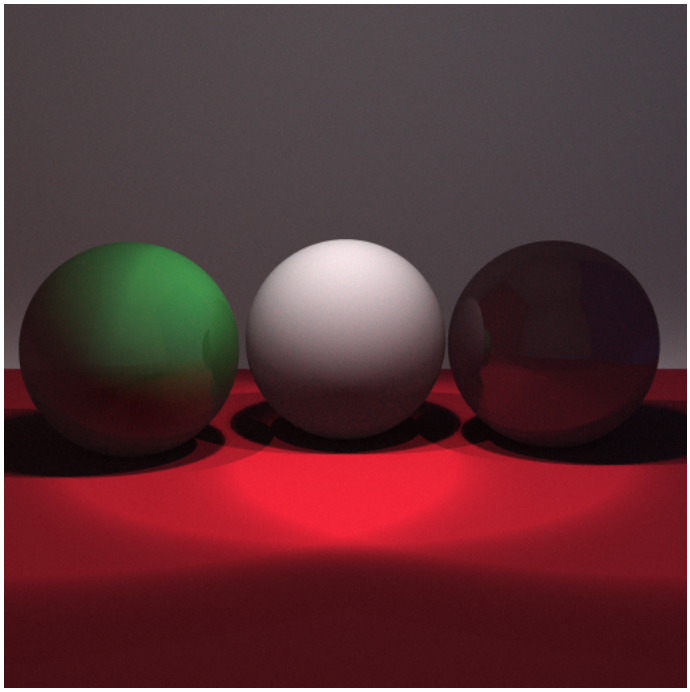


Figure 5: Three spheres illuminated by three spotlights, each light with ten degrees of fall off.

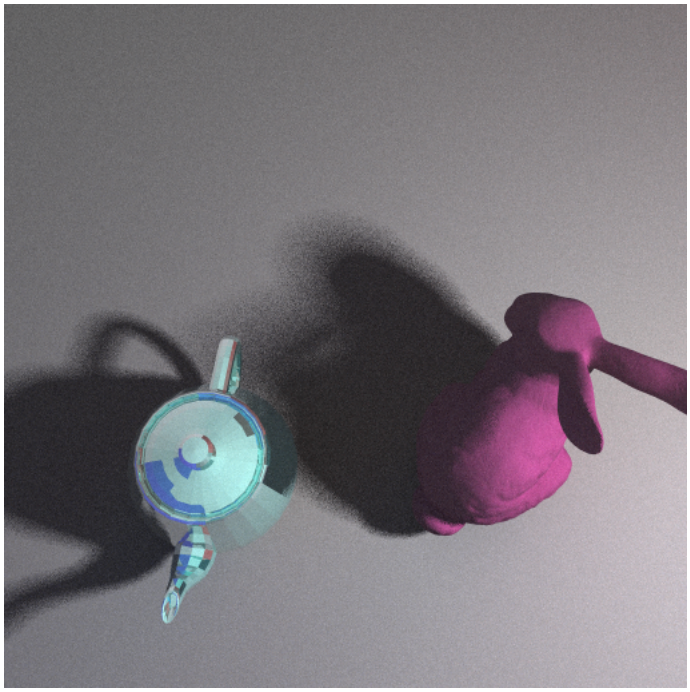


Figure 6: Teal metallic teapot and matte pink bunny inside Cornell Box, illuminated by spherical area light.

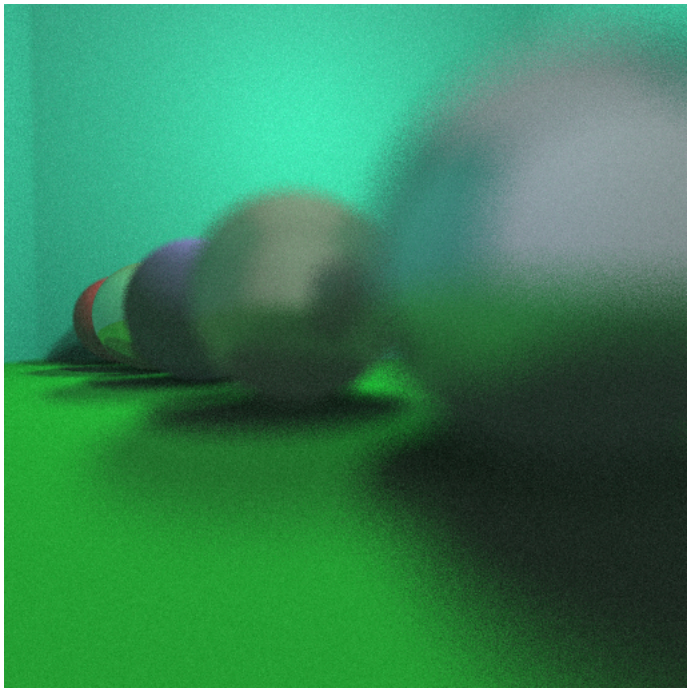


Figure 7: Line of spheres: Focal depth at z-axis = 50.0

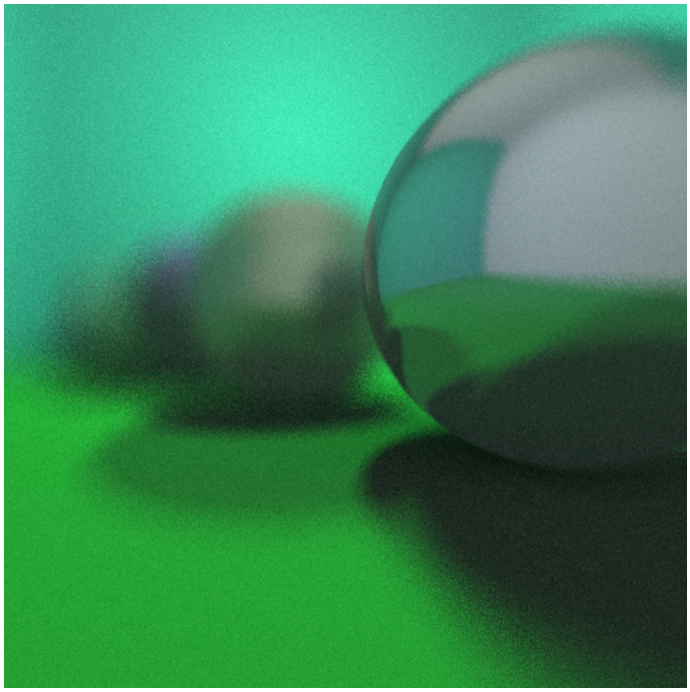


Figure 8: Line of spheres: Focal depth at z-axis = 140.0

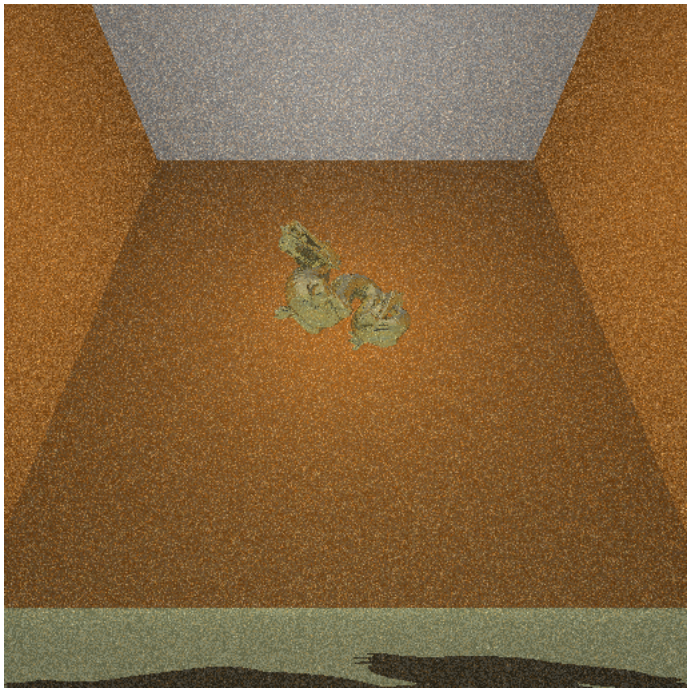


Figure 9: Floating dragon, 1 sample per pixel

# 8 Future Work

Extending features of my unidirectional path tracer has shown me areas for improvement:

- **Cleaner C++ style:** My path tracer implementation is a result of refactoring and extending a past JavaScript project into C, which I later extended with C++. Each language carried different conventions that have either lingered or have been unresolved. For example, I was already familiar with memory management in C, which is done through `malloc` and `free`. However, I was less familiar with memory management for C++ classes and exactly what parts of memory are cleaned up by their automatic destructors. I also used a mix of C++ style `new` along with C style memory management. The path tracer code contains a mix of imperative and object-oriented style code. Ideally, the code base should be refactored to stick to a pure object-oriented approach.
- **Algorithm refinements:** My path tracer uses simple but accurate approaches like unidirectional path tracing and uniform sampling. To make the program more efficient, bidirectional path tracing could be implemented, as well as better sampling methods such as importance sampling.
- **Parallelization:** Beyond the main render loop, other areas of the code could be parallelized. An area that was a rendering bottleneck was the construction of a kd-tree for acceleration, which was built serially. An idea for parallelization would be to parallelize the construction of the subtrees. Children nodes of the same parent are independent from one another and their construction should be able to occur simultaneously.

The suggestions above specifically point out extensions to pre-existing features in my path tracer, but there are also a variety of other features that could be implemented for allowing a richer visual expression, such as texture mapping, volumetrics, etc.

# 9 Acknowledgments

Thank you to Professor Holly Rushmeier for advising me in my independent studies for three semesters. Like last semester, I would also like to thank my former classmates Brian Li and Jakub Kowalik for pursuing studies in computer graphics with me. Thank you to my friend Karl Li, who always inspires me to learn more.

# References


[1] Tomas Akenine-Moller. Triangle-box intersection testing. http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/. Accessed: 2017-04-04.

[2] Keith Lantz. Surface area heuristic for kd trees. http://www.keithlantz.net/2013/04/kd-tree-construction-using-the-surface-area-heuristic-stack-based-traversal-and-the-hyperplane-separation-theorem/. Accessed: 2017-05-03.

[3] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.