Monmouth College

Capstone Proposal - Ray Tracing

Ryan McKirdy

Dr. Logan Mayfield, Dr. Robert Utterback

COMP 401

19 November 2020

**Abstract**

This project aims to construct a ray traced image renderer from scratch using mathematical principles. Inspired by homegrown approaches to ray tracing and supported by a semester of research on the extensibility of mathematics for the use in computer graphics, this exploration of ray tracing aims to address the relationship between performance and complexity. Measurable progress will be demonstrated in two week intervals, with a prototype featuring new advancements to be showcased at the end of each interval. Stages of progress will include transitioning to a 3D image renderer, adding material varieties, and implementing aspects of ray tracing. Code optimizations will be made at the end of each stage in hopes of decreasing render time and resource usage. The final product will be a fully featured ray tracer capable of rendering arbitrary or predefined scenes composed of a range of objects and material definitions. Data on performance metrics will be collected as the complexity of the ray tracer increases, and analyzed to show the relation between complexity and computational intensity. Rendered images will be curated and organized into portfolios showing the evolution and improvement of the ray tracer.

**Background and Introduction**

Ray tracing is considered by many to be the pinnacle of computer graphics. At its core, ray tracing refers to the mathematical simulation of how light bounces, and its use in computer graphics. Due to its high computational expense, computer graphics traditionally have used ‘rasterization’ for performance sensitive applications.

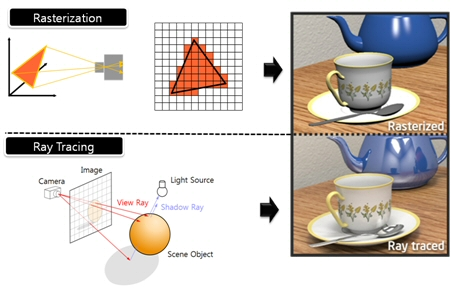


Figure 1: Visual explanation of the difference between traditional rasterization and ray tracing

<http://mvlsi.kaist.ac.kr/research/multimedia-processor/ray-tracing>

The difference in time spent rendering the same scene in rasterization versus in ray tracing can be magnitudes apart. For the most part, rasterized computer graphics have advanced to the point where they can display great complexity while remaining performance sensitive. Only in recent years has ray tracing been able to begin catching up to the real-time applications. So, the main problem this project attempts to address is: How robust can a ray traced image renderer coded from scratch be, and how performant can it become? This question will be addressed by incrementally building up the complexity of the image renderer and ray tracer, from a simple scene like Fig. 2 below, to a more complex scene like in Fig. 1 above.

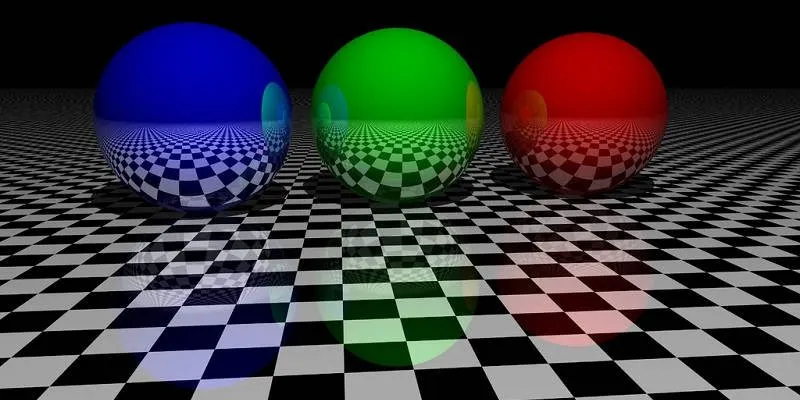


Figure 2: A simple ray traced scene demonstrating reflectivity

<https://www.maketecheasier.com/what-is-ray-tracing/>

As the complexity builds, data on performance metrics will be collected during each render and averaged as a whole, in hopes of painting a better picture of how render times increase with computational intensity, while also attempting to minimize increases in render time with code optimizations. A unique aspect of this project is its goal of analyzing render times throughout its evolution. Beginning as a simple 2D rasterizer, becoming a 3D rasterizer, and finally ending up as a fully featured 3D ray tracer, there will be plenty of data available to analyze throughout its transition as the line between rasterization and ray tracing becomes less apparent.

Inspiration for this project was found in an analysis of one of the world’s most succinct ray tracers. A man named Andrew Kensler coded a C++ based ray tracer so compact that it would fit on the back of a business card, using a mere 1,337 bytes (Sanglard). Through an analysis of the expanded and translated code, it becomes apparent that the code does indeed represent a fully featured ray tracing application. In a similar vein, this project aims to maximize performance, rather than to minimize code size.

**Project Description and Analysis**

The completed project can be divided into three categories: A C++ based ray tracer, Python based performance analysis, and a portfolio of images and data produced throughout the project.

The C++ ray tracer will output pre-rendered 3D image files of variable resolution, and incorporates a number of features. It will produce fully ray traced scenes, with global illumination, reflections and soft shadows. Scenes can consist of basic geometric objects, as well as more complicated importable meshes. These scenes can be randomly generated, or predefined. The ray tracing system will support a range of different materials, including reflective, refractive, emissive and light absorbing properties. Along with each render, data metrics on render time, CPU usage, and memory usage will be appended to a spreadsheet for use in performance analysis.

The Python based performance analysis will use data on rendering performance, to be collected over the duration of the project. After completing each stage in the project, renders will tag a spreadsheet with relevant performance data, creating a database to be averaged and analyzed. Variables like image resolution will remain under constant scrutiny throughout all stages, while vertex/face counts or ray bounce depth will only be applicable to some. Optimizations will also be made to the ray tracer at each stage in the project, and the performance difference will be measured as well. Render time will be the main metric for analysis, but data will also be gathered on memory and CPU usage.

The portfolio will consist of an art gallery of the most interesting images rendered at each stage of the project, as the ray tracer becomes more robust. Most will be generated using the random scene generator, but some will showcase a predefined scene and its visual evolution as the project evolves.

**Foundations**

First, an investigation of the mathematics behind ray tracing was needed to determine the feasibility of coding each aspect. Many of the mathematical principles behind ray tracing have to do with vectors and surface normals. The ‘normal’ of a surface refers to the vector that is perpendicular to a certain polygonal face. This single piece of data, when combined with a location in three dimensional space, is very powerful.



Figure 3: Illustration of tracing for a single pixel

Fig. 3 illustrates the process for the ray tracing of a pixel, from light source to object to camera. Any ray from the light source that contacts the object will do so at a certain angle relative to the normal vector at the location of intersection. Thus, the angle that the ray is deflected at off of the object can be calculated using that angle. The deflected ray either bounces again or hits the camera, yielding a single complete ray trace. In essence, this process uses only vector math, algebra, and trigonometry, all mathematical concepts that are easily demonstrable in most any programming language.

Once a basic level feasibility of the project had been determined as being possible, a demonstration of the math applied in programming was needed. Beyond mathematics, the backbone of the project lay in developing an efficient and simple image renderer that would be extensible to more complex work later on. This is where the bulk of the applied foundational research was done. It was determined that an ideal demonstration would be to create a 2D image renderer capable of outputting a randomly generated scene. First, a method of rendering images had to be established. The PPM (Portable Pixel Map) format provided an easy way of codifying images, since the image format standard consists simply of ASCII text formatted into a header and pixel values.



Figure 4: Depiction of the pixel rasterization process

As shown in Fig. 4, the origin of the image is the top left. By looping through the pixels, first by each pixel in a row and then by each row in the image, a complete image can be constructed. From here, classes of objects (rectangles and circles) were established, each containing coordinates and RGB values. Now, the image is able to be mathematically iterated based on equations for rendering the region of each rectangle or square. If a pixel is within the confines of an object, its color values are changed to that of the object. Randomization was established to provide the random scene generation, and applied to the colors, origins, and sizes of all objects. The 3D image renderer will function in a similar form.

At such an early stage in the project, there are many open problems to be considered. The 3D image renderer will not be a difficult component of the project to establish, as most of the two dimensional mathematical thinking only requires the addition of the third dimension to equations in order to function properly. More challenging components will come into play when pixel sampling and ray tracing mathematics are added, as these require more abstract mathematical formulas to execute. Some of the ray traced materials, like transparency, will prove more difficult to implement as well, since they require additional sets of vector math to return a pixel’s color value beyond a simpler reflective material. Rays need to be scattered into the object as well as off of it, and the values returned by each direction need to be blended while still accounting for further bounces of outbound rays. Additionally, research will need to be conducted on the rendering polygons as flat versus smooth shaded. While these are all open problems, a solid mathematical explanation exists for each of them, lending a high degree of confidence that they are easily solvable in code.

**Implementation Plan and Timeline**

All C++ coding and compilation will be done in Microsoft Visual Studio. At the time of writing, the libraries ‘cmath’, ‘fstream’, ‘vector’, and ‘stdlib’ are in use. Python analysis of the data will be conducted using Jupyter Notebook under an Anaconda environment, for the productive benefits in interactivity of data visualization it provides. Once a stage of the project has been completed, compilation of performance data for that stage can begin. Using random scene generation, 100 runs of the renderer will be done, each of which will output performance data and rendering time into an Excel file, tagged with the relevant stage in the project. At the end of the project, all gathered data will be analyzed using Python, with the libraries Pandas (Pandas Development Team) and Matplotlib (Hunter). Results will be aggregated on a per stage basis as well as for a complete scope of the project. Additionally, GitHub repositories will be used for versioning control of both the C++ and Python sides of the project

Throughout the course of the 15 week semester, milestone progress will be made on two week intervals, as shown below. The vector math and 3D image renderer stages will be completed over the two month winter break period in order to avoid an extended period of not making any progress.

|  |  |
| --- | --- |
| **2D Image Renderer** | Completed |
| **Vector Math** | Over winter break |
| **3D Image Renderer** | Over winter break |
| **Material System** | Week 2 |
| **Depth Testing and Sampling** | Week 4 |
| **Ray Tracing - Diffuse** | Week 6 |
| **Ray Tracing - Shading** | Week 8 |
| **Ray Tracing - Materials** | Week 10 |
| **Polish and Bugfix** | Week 12 |
| **Final Preparation** | Week 14 |

For the ‘2D Image Renderer’ stage, in addition to the C++ Vector library, other vector based mathematical operations like dot and cross products will be defined to aid in future intersection and ray casting operations. The coordinate based object formats utilized in the previously completed 2D image renderer will be converted to incorporate the new vector math. At the end of this stage, the 2D image renderer will be updated to allow tracing of shaded regions. Any random 2D scene will contain a light source, and vectors traced from that light source to the edges of objects will dictate where shaded regions occluded by the objects will lie, as illustrated in Fig. 5:

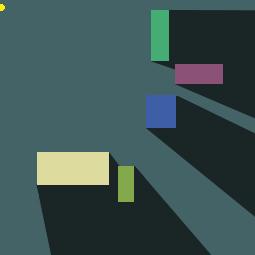


Figure 5: Visualization of using vectors to trace lit and shaded regions of a 2D image

Following the establishment and integration of vector math, progress on the ‘3D Image Renderer’ stage can be made. This stage will allow support for a variety of conventional two-dimensional and three-dimensional objects (Planes, cubes, circles, cylinders, spheres, and cones) as well as support for arbitrary 3D meshes, using the PLY (Polygon File Format) object format (Bourke). Like the PPM image format used to render images, the PLY format is comparable in its simplicity, consisting of an ASCII representation of properties, vertices, and faces.

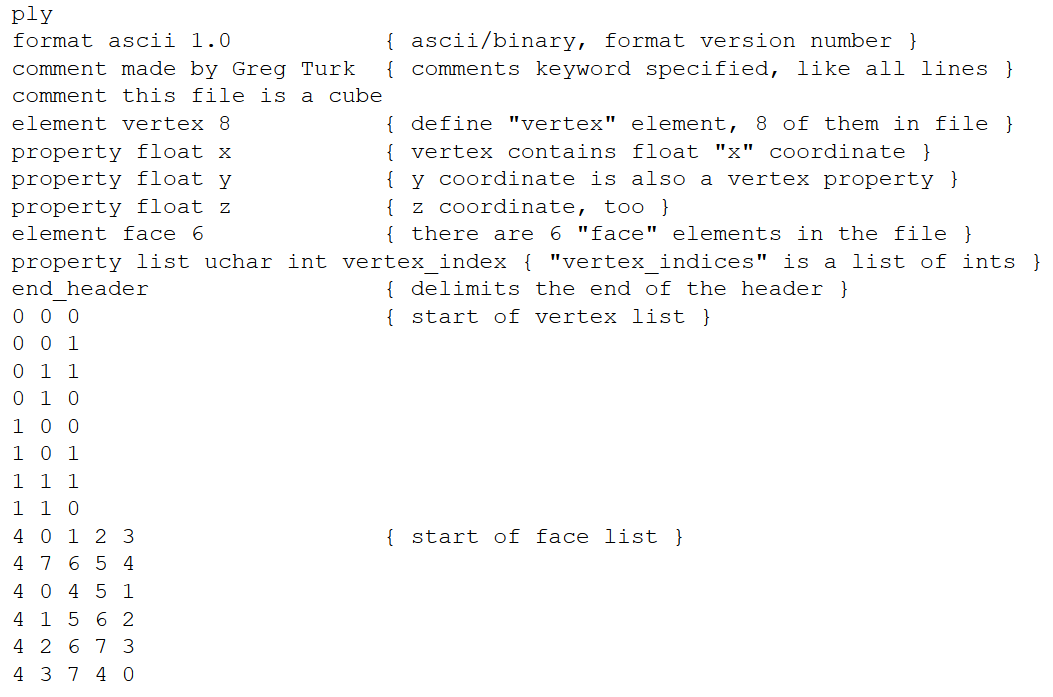


Figure 6: The text of a .ply file containing a cube (Bourke)

In the event that additional functionality is needed, there are a variety of C++ libraries for interfacing with PLY files available, such as ‘TinyPly’ (ddiakopoulos). In the coming stages, support for the convex simple meshes will be prioritized ahead of the potentially concave arbitrary meshes. A ray drawn through a convex mesh has one entry point and one exit point, whereas a concave mesh may have multiple entry and exit points. This may serve to complicate the ray tracing process, and will only be worked on in later stages if time permits. Completion of the vector math and 3D image renderer stages ahead of the start of the semester will provide a solid basis upon which to begin building the fundamentals of ray tracing when the semester begins.

The ‘Material Definitions’ stage will consist of defining various materials to be used within the ray tracing system. Since no actual ray tracing will be implemented at this stage, most will simply be placeholders to be implemented in later stages. It will be easier to establish the material system in advance of implementing ray tracing for code structure purposes, which is why this stage is scheduled for completion in advance of the ray tracing stages. In total, the system will support seven different types of materials that will be able to be combined with each other arbitrarily. They are as follows:

|  |  |
| --- | --- |
| **Diffuse** | The color of an object, in RGB values. |
| **Texture** | The color of an object, determined by an image texture. Texture mapping will be done using simple UV mapping for each object type. |
| **Reflectivity** | The amount of light an object reflects. A fully reflective object would be a perfect mirror. |
| **Roughness** | The diffusion of light across a reflective object. High values scatter reflections farther across a reflective object, low values are more mirror like. Illustrated in Fig. 7.    Figure 7: Rendering of the relation between reflectivity and roughness  <https://commons.wikimedia.org/wiki/File:Specularity-phong-showcase.png> |
| **Transparence** | How much light passes through an object, and at what angle it is refracted. An object with no index of refraction (IOR) is simply see through, one with a high IOR looks like glass. Illustrated in Fig. 8.    Figure 8: Mathematical illustration of the Index of Refraction for water  <http://www.math-principles.com/2014/10/index-of-refraction-problems.html> |
| **Emission** | How much light an object gives out. Light color is inherited from diffuse or texture color values. |
| **Subsurface Scattering** | Amount of light absorbed by the surface of an object, which is then scattered throughout its body. Scattering color is mixed with object color values. |

Since ray tracing will not be implemented at this stage, only the diffuse, texture, and reflectivity material systems will be implemented. The remaining material systems will be implemented in the ‘Ray Tracing - Materials’ stage. Since reflectivity is not yet able to be implemented in an actual ray tracing capacity, a visualization of its future effect will be accomplished by using its values, as well as math that will be useful later, to create an approximation for each polygonal face. At the end of this stage, the finished reflectivity and color models will be implemented into the 3D image renderer produced over winter break, and another portfolio of images will be produced.

The ‘Depth Testing and Sampling’ stage will establish crucial principles in preparation for ray tracing. Rays cast throughout the three dimensional scene will be able to sample what type of object they are intersecting with at each bounce, and receive the relevant material and surface properties for each to be communicated forward to the next bounce or termination. Depth testing will also be useful later on in material definitions like subsurface scattering that rely on object dimensions for their visualization, or to control the intensity of an emissive material, and properly attenuate the distance its light is scattered into a scene. To control the distance that light scatters into a cube with subsurface scattering, for instance, its dimensions must be known so that an incoming ray can be terminated at an appropriate distance inside the object. Sampling is the rasterization-like process of determining a final pixel color value based on a completed ray trace. Any given pixel needs to be repeatedly sampled to average and anti-alias the values of adjacent ray traces. At the end of this stage, the 3D image renderer will be converted to provide a visualization of depth in scenes, with white being the foreground, and back being the background. While not directly relevant to creating a ray tracer in a visual regard, creating this functional prototype of depth visualization will be a useful debug tool for verifying that everything in the scene is being rendered as it should be, and may help to fix visual bugs in the final stages of the project.

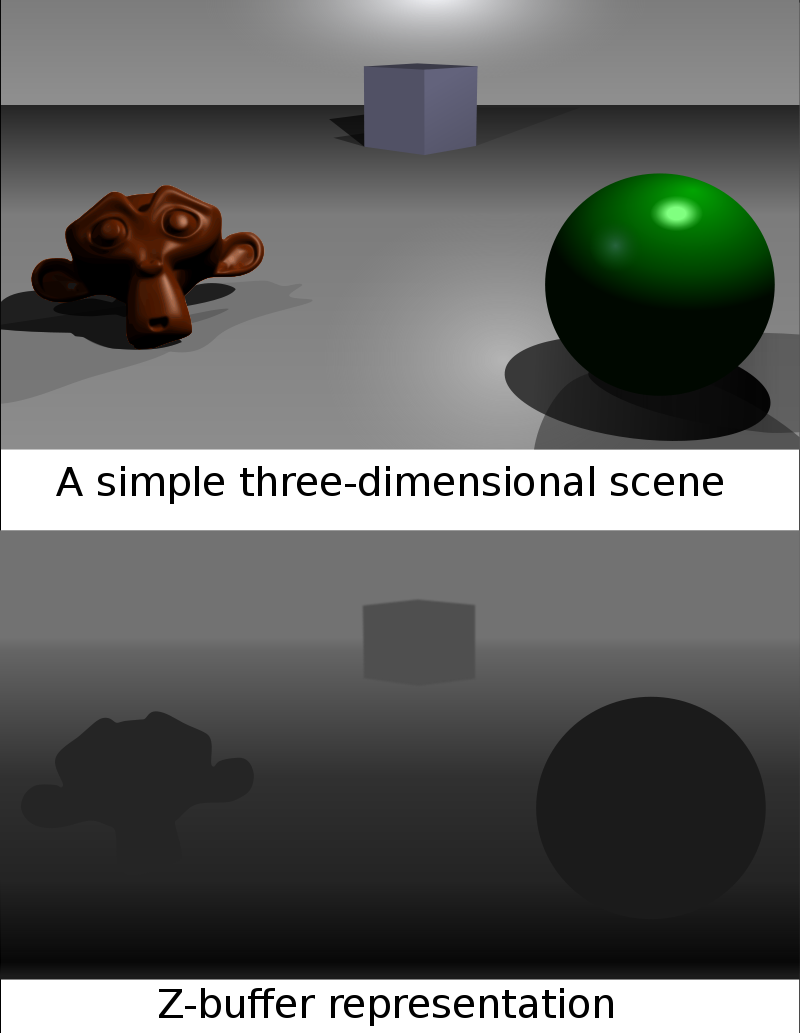
****

Figure 9: Representation of the computation of depth in a 3D scene

<https://en.wikipedia.org/wiki/Z-buffering>

The ‘Ray Tracing - Diffuse’ stage will see the first implementations of practical ray tracing. Only color data will be taken into account at this stage, to simplify the process of defining the underlying math of casting rays. The prototype at this stage will integrate into the 3D image renderer, and produce results similar to the rasterized images previously generated. Ray bounces are set up to occur, but are terminated after the first bounce.

In the ‘Ray Tracing - Shading’ stage, the proper visualization of ray tracing will finally be seen. The color data established in the previous stage will now bounce and illuminate the scene as it should, and shadows and ambient light occlusion will provide a realistic simulation of light. Depth sampling set up in previous stages will be useful for gathering data on how far a shadow’s rendered edge should disperse, for the implementation of soft shadows that mimic real life. Once again, the prototype for this stage will integrate into the 3D renderer.

For the ‘Ray Tracing - Materials’ stage, with ray tracing of the scene now in place, the placeholder work accomplished in the ‘Material system’ stage can be properly implemented. All seven materials described previously will be implemented. In the event that the project is behind schedule, the transparence and subsurface scattering materials will be the last to be implemented. More advanced vector math and sampling are needed to properly implement them, so they will be the most time consuming to implement. The prototype at this stage can be considered ‘fully featured’ in the scope of this project, and will incorporate all the major aspects of ray tracing into the 3D image renderer. Generated scenes will be fully shaded, and all materials will interact with each other as they should.

The ‘Polish and Bugfix’ stage will be reserved for addressing any leftover bugs that have accumulated throughout completion of the stages. Due to the complexity of ray tracing math that will have accumulated by this stage, it is likely there will be visual artefacts and glitches in renderings that will need to be addressed. For this stage, an illustration of the differences that fixing bugs makes and the impact on image quality will be shown.

During the final preparation stage, any final adjustments that need to be made to the ray tracer will be done. Final papers and presentations will be completed, and portfolios of the best rendered images from each stage in the process will be compiled, as well as final performance analysis data.

In addition to each stage in the process, an optimization for each stage will be implemented in an attempt to cut down on render time and resource usage. The attempted optimization for each stage’s prototype will be as follows:

|  |  |
| --- | --- |
| **2D shaded regions** | Only do shading for unshaded regions, eliminating re-computation for overlapping regions |
| **3D image renderer** | Render from foreground to background, eliminating re-rendering of obscured pixels |
| **Reflectivity material visualization** | Adjust number of polygons in meshes to decrease shading workload |
| **Depth visualization** | Implement culling for non-visible faces, hastening render times |
| **Diffuse ray tracing** | Adjust ray casting system and pixel sampling, optimizing while remaining conscious of image quality |
| **Shaded ray tracing** | Refactor tracing mathematics to be less computationally intensive |
| **Material system ray tracing** | Adjust ray bounce depth relative to perceived image quality |
| **Polish and Bugfix** | Refactor material system vector math |

**Conclusion**

To summarize, this project is simultaneously an exploration of coding a ray tracer from scratch, while analyzing its performance relative to its complexity. Features like complex object rendering and varied material systems will make for a robust ray tracer with ample room for performance optimization. At its final stage, the project will include a fully featured C++ ray tracer, a suite of Python data analysis tools illustrating the relationship between code complexity and performance, and a portfolio of images showcasing some of the best images generated for each stage of the project. This will answer the proposed question of ‘How robust can a ray traced image renderer coded from scratch be, and how performant can it become’.

The capstone to end an entire college career should be an exploration of something that resonates with you, and for me this exploration of ray tracing and mathematics accomplishes just that. Computer graphics and ray tracing have always fascinated me, and my experiences with coding this project from the ground up have proven to be very rewarding thus far. The problem solving experience of implementing each progressive stage of mathematics and code will surely provide the same satisfaction, as will watching the artistic progression of the produced images. With proper adherence to the stages and deadlines defined in this proposal, this project will make for a fulfilling and fruitful capstone experience in the coming semester.

**Bibliography and Works Cited**

Balajadia, Armando Magtalas. “Index of Refraction Problems.” Math Principles, 11 Oct. 2014,

[www.math-principles.com/2014/10/index-of-refraction-problems.html](http://www.math-principles.com/2014/10/index-of-refraction-problems.html).

Bourke, Paul. *PLY - Polygon File Format*, [www.paulbourke.net/dataformats/ply/](http://www.paulbourke.net/dataformats/ply/).

ddiakopoulos. “Tinyply 2.3.” *GitHub,* 27 July 2020, [www.github.com/ddiakopoulos/tinyply](http://www.github.com/ddiakopoulos/tinyply).

Hunter, John, et al. “API Overview.” *Matplotlib 3.3.3*, Matplotlib Development Team, 12 Nov. 2020,

[www.matplotlib.org/api/index.html](http://www.matplotlib.org/api/index.html).

Pandas Development Team. “API Reference.” Pandas 1.1.4 Documentation, 2020,

[www.pandas.pydata.org/docs/reference/index.html](http://www.pandas.pydata.org/docs/reference/index.html).

“Phong Illumination Model.” *Wikimedia Commons*, Wikimedia Foundation, 4 Oct. 2020,

[www.commons.wikimedia.org/wiki/Category:Phong\_illumination\_model](http://www.commons.wikimedia.org/wiki/Category:Phong_illumination_model).

“Ray Tracing.” Multimedia VLSI Laboratory,

[www.mvlsi.kaist.ac.kr/research/multimedia-processor/ray-tracing](http://www.mvlsi.kaist.ac.kr/research/multimedia-processor/ray-tracing).

Sanglard, Fabien. “Fabien Sanglard's Website.” *Fabien Sanglard*, 20 Sept. 2013,

[www.fabiensanglard.net/rayTracing\_back\_of\_business\_card/index.php](http://www.fabiensanglard.net/rayTracing_back_of_business_card/index.php).

Stobing, Chris. “What Is Ray Tracing and Why Is It the Next Big Thing in Gaming Technology.”

*MakeTechEasier*, Uqnic Network Pte Ltd, 30 Aug. 2018, [www.maketecheasier.com/what-is-ray-tracing/](http://www.maketecheasier.com/what-is-ray-tracing/).

“The C++ Resources Network.” Cplusplus.com, 2020, [www.cplusplus.com/](http://www.cplusplus.com/).

“Z-Buffering.” *Wikipedia*, Wikimedia Foundation, 28 Oct. 2020, [www.en.wikipedia.org/wiki/Z-buffering](http://www.en.wikipedia.org/wiki/Z-buffering).