**Final Project:**

**Raytracing, Edge Silhouettes, and Cel-Shading**

CSE 6413 - Computer Graphics - Fall 2019

Dalton Pruitt – dp987

December 12, 2019



Contents

[Initial Thoughts 3](#_Toc27078032)

[Development Path 4](#_Toc27078033)

[Technical Background 5](#_Toc27078034)

[Ray Tracing 5](#_Toc27078035)

[Edge Silhouettes 9](#_Toc27078036)

[Cel-Shading 9](#_Toc27078037)

[Example Images from Finished Project (before presentation) 10](#_Toc27078038)

[Problems Fixed Since Presentation (December 6th) 11](#_Toc27078039)

[Documentation of Project Code 14](#_Toc27078040)

[Noteworthy Data from Project 17](#_Toc27078041)

[Personal Takeaways 17](#_Toc27078042)

[Future Considerations for this Project 17](#_Toc27078043)

[References 19](#_Toc27078044)

# Initial Thoughts

After reading through Burnett and Piazza’s report, I decided that I wanted to implement cel-shading, namely in respect to the design of two of my favorite games from the *Legend of Zelda* series: *The Wind Waker* and *The Breath of the Wild*. The cartoony look is appealing to many people, and I wanted to emulate it. However, as cel-shading is a very basic shading procedure, the real brunt of the work for this project is composed of the ray-tracing component, which is used to give reflections in reflective objects in the scene. This was also inspired by the report by Burnett and Piazza, and a picture from the paper is shown below.

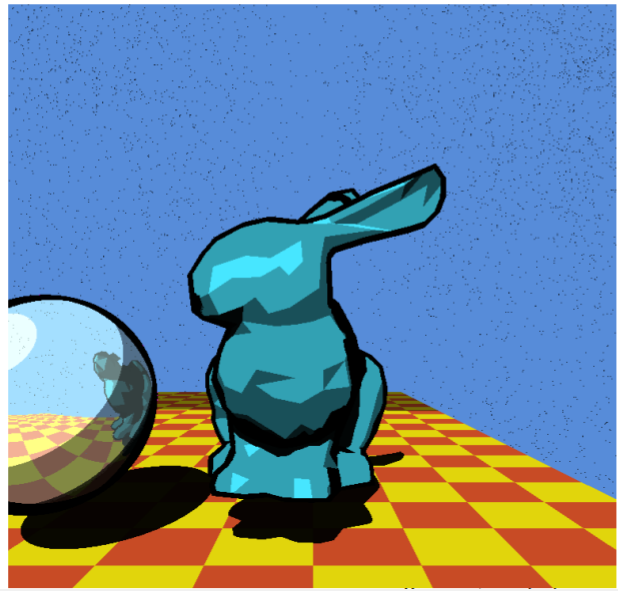


Figure 1. Image from Burnett and Piazza [1]

Implementing a ray tracer in JavaScript and drawn using the regl wrapper for WebGL (and on most other platforms) would be slow, due partly to the possible recursive nature of the algorithm (if performing more than one reflection) and due mostly to the sheer number of rays cast and objects that have to be checked for intersection (at least one for each pixel). The total runtime could be O(# of initial rays \* # of triangles/objects \* # of bounces), which is incredibly large for high-resolution pictures. Proper “spatial acceleration” data structures can be used to improve the time taken during the intersection-searching, and the rays can be cast in parallel by using frameworks for GPU programming such as the NVIDIA CUDA language and the NVIDIA OptiX framework, also mentioned in the Burnett/Piazza report.

Given my familiarity with the Python language, and the fact that ray tracer’s have been built in Python prior, I decided to create a ray tracer in Python. Following a production goal similar to one of the aforementioned Python ray tracers, my main output would be PNG files created using the Pillow library. The individual pixels will each have a ray sent out, tested for intersection against geometry, then bounced and the process repeated until the maximum number of bounces has been reached or the object hit is non-reflective. The image is completed once every pixel has undergone this process.

# Development Path

My development path prior to the presentation on December 6th was as follows:

1. Get a Python project set up in a development environment (PyCharm)
2. Following the same image-creation procedure in [2], set up PNG output (checkerboard pictures)
3. Create Vector3, Ray, and Triangle classes. These are “original” in the sense that I did not look at other sources for “inspiration” (which made me lose time debugging)
4. Render basic triangle
5. Render reflective triangle
6. Render two reflective triangles (blue/green and blue/blue pictures)
7. Import and render 3D model (5860 triangles)
8. Add Sphere class (to test lighting)
9. Add Phong shading model to ray-tracing, test with spheres
10. Add a basic transform structure to allow models to face the correct way; render model facing camera
11. Add edge silhouettes, render spheres
12. Add cel-shading, render spheres (1024x1024 image shown in class presentation)

Following the presentation:

1. Add in cube from cube\_regl.js demo file.
2. Fix issue discussed in presentation: edge silhouettes not working with triangles
3. Add spatial acceleration structures (Bounding boxes) to reduce computation time
4. Render 3D model with cel-shading, edge silhouettes, and reflections

# Technical Background

This section details the relevant technical details surrounding the implementation of the three main implementation goals of this project, in order of decreasing complexity: ray tracing, edge silhouettes, and cel-shading.

## Ray Tracing

As was discussed in numerous presentations in class on December 6th, ray tracing is a popular global illumination model. While most of the rendering students have been *object-order rendering,* where the objects are dealt with and their effects on pixels result in the final image, ray tracing is an inherently *image-ordered* rendering technique, where each pixel is dealt with and the objects that affect the pixel are found and their effects on the pixel calculated. [3]

The main goal of ray tracing is to find for each pixel what object is seen by that pixel. This is accomplished by using the mathematical notion of a “ray”, that is, an object in three-dimensional space with a start point, or *origin*, and a direction. The location of the camera/eye is used as the origin for every ray (in the basic case, at least), and the pixel’s location on a two-dimensional grid offset from the ray’s origin is used in the following calculation:

The ray is then tested against all geometry present in the “scene” to be rendered (whether or not is actually in front of the camera) to determine whether or not the ray intersects any object. The various “ray-intersects-<object>” tests will be explained in this section. Figure 1 below is a popular image helpful for visualizing ray tracing.

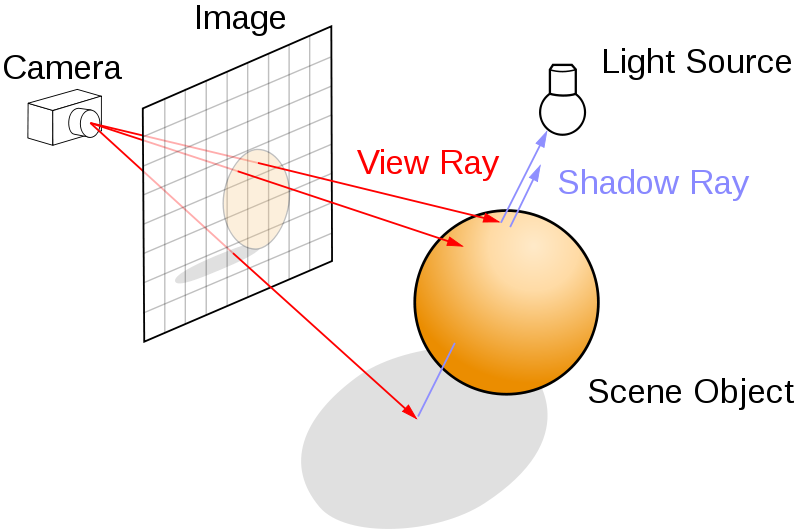


Figure 2. Ray tracing visual aid [4]

If the ray does not intersect an object, the routine can simply save the background color to the current pixel. If the ray does intersect an object, and it is known which object the ray intersected first, i.e. the object with the smallest “distance to hit,” then the pixel can be colored depending on the shading of that object at the point that was hit, including specular interactions with other objects. I included reflections, as that was the main goal for the use of raytracing in the first place. However, shadows are a nearly-free addition (at lease code-wise) to a ray tracer, as one just needs to use “shadow rays” from the current point hit to the light sources and check for any geometry in the way to determine if the current point is in shadow.

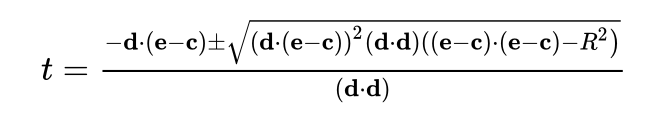
To perform reflections, the normal of the point hit must be available (or able to be calculated) and is used in the following formula to get the reflection direction:

One may notice that this form of the equation is the negative of that present in the calculation of that is present in the Phong shading model. This is due to the incoming ray’s direction vector having a negative dot product with the object’s normal at the point. This reflection direction and the position of the point hit can be taken as the direction and position of a reflected ray, which can then be used to determine what specular interactions exist for the given point within the rest of the scene.

The process of sending a ray from the camera to the pixel, checking for intersection, and reflecting off of it can be defined recursively, as the new ray will need too be checked for intersection against geometry and shaded if hitting something. This forms the basis for the entire “raytrace” function used in the program.

As mentioned above, different “ray-intersects-<object>” tests exist for various types of objects, such as spheres, triangles, general polygons, planes, and bounding boxes. The main goal of these checks is to obtain the identity and position of the closest point to the origin of the ray that the ray intersects (ignoring intersections behind the origin). The distance to an intersection is calculated as a multiple of the length of the direction vector, which if normalized is 1, and is denoted by a “t”. If an intersection occurs, t is saved as the ray’s “closest/nearest hit distance” and is used as the maximum bounds in further intersection checks. Typically, is initialized to some extraordinarily large quantity to denote no intersections.

The arguably “easiest” one of the intersection, at least in my opinion, is the “ray-intersects-sphere” check. It can be found in the text by Shirley and Marschner, chapter 4, and is as follows: [3]



Where:

t = distance to intersection

d = ray direction (normalized)

e = ray origin point

c = center of sphere

R = radius of sphere

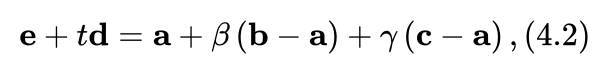
When the normal is required at points of intersection, it can be readily calculated by the equation:

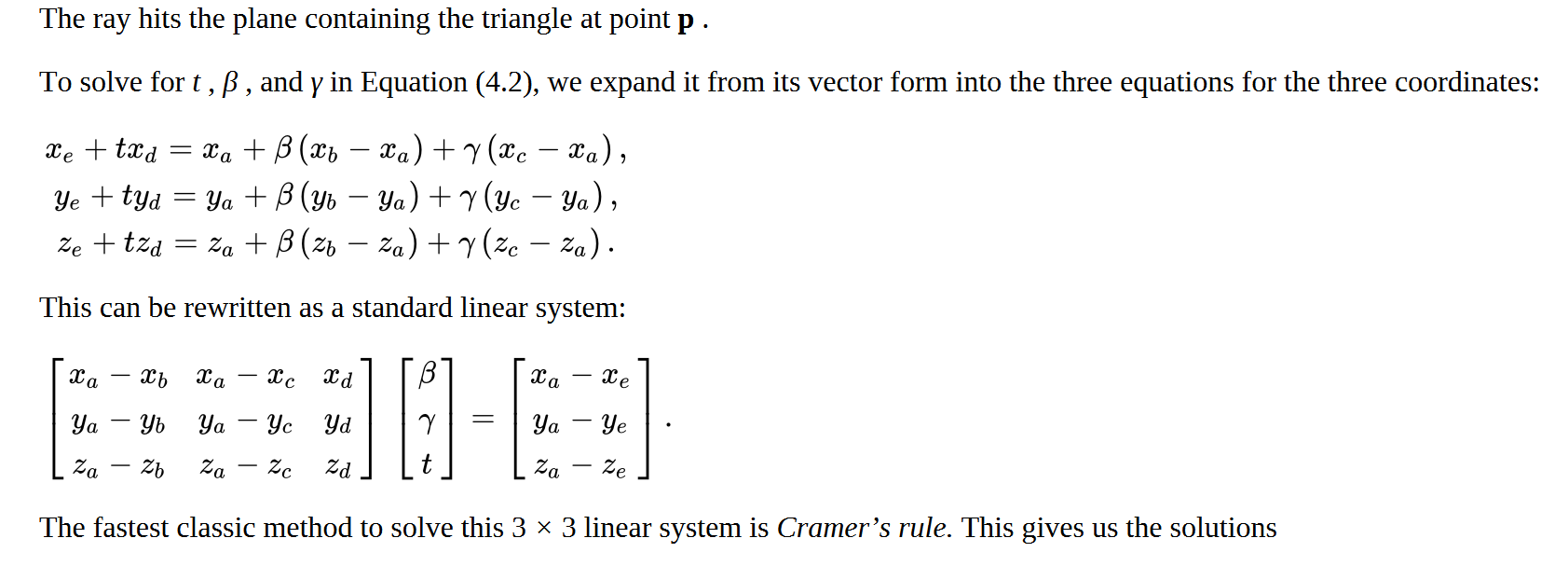
And the hit position can be calculated by:

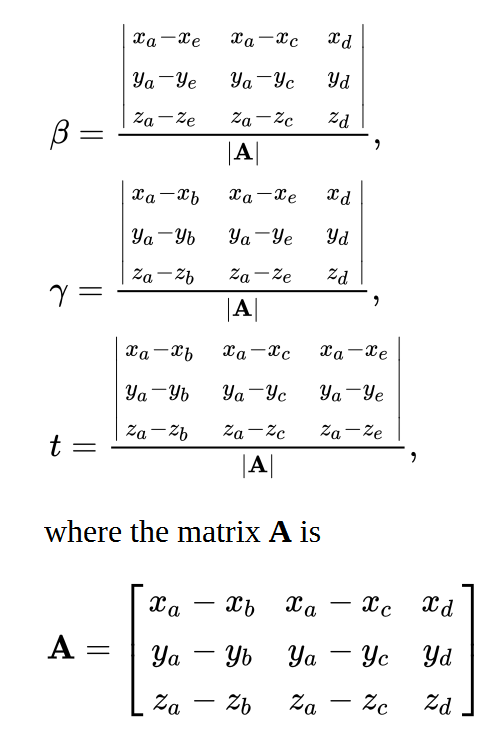
This last equation is applicable to other types of objects, as well, due to not differentiating between types of objects.

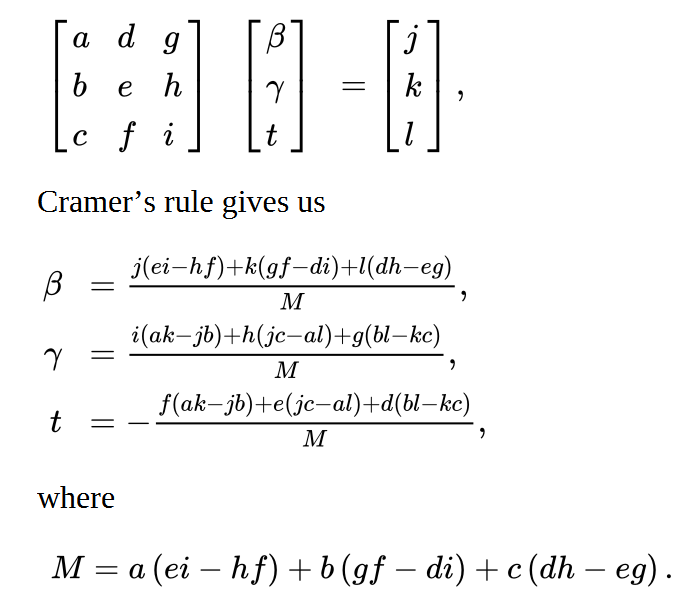
Ray-triangle intersections are among the most complex calculations, but the same text contains a method and even pseudocode for such interactions, and the relevant excerpts are given below. [3]

The math:

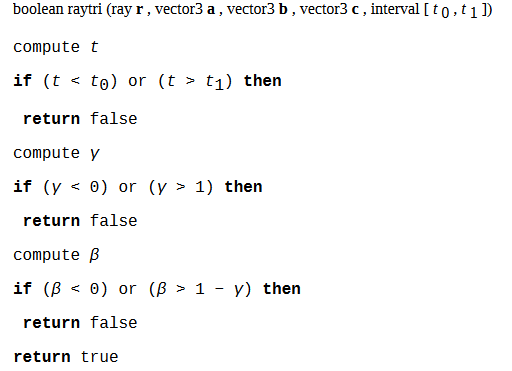








The pseudocode:



I chose to include these pictures as I do not fully understand how the math works, due to the length of time since I have taken Linear Algebra, but I still used these calculations and the structure of the pseudocode in my triangle-intersection routine. All credit given to the source, of course. [3]

Finally, ray-bounding box intersections was the last intersection type implemented. The routine used was based on a webpage by ScratchAPixel [5] which was in turn based off of an article by Amy Williams, et. al. [6] The main goal is to make sure that the ray has a section that is within the bounds of the 6 planes that define the bounding box. The routine had to be modified to allow for the returning of a reference to the object intersected, since the bounding boxes were defined in such a way as to hold either two bounding boxes or a single triangle. So, if the outer bounding box is hit, either the two contained bounding boxes need to be checked for intersection or the contained triangle needs to be checked for intersection. As such, this intersection process is defined recursively.

## Edge Silhouettes

The addition of edge silhouettes can typically be readily achieved through pre-defined means, such as by using the G-buffer in OpenGL/WebGL. In this case, since I was already using raytracing, I decided to use more rays (spread out a bit) for each pixel and determine the hit object for each ray. If the rays hit the same object, then shade that object (with less spread-out rays). If none of the rays hit an object, use the background color. If the rays hit different objects or an object and no object, color black for the silhouette. This made it a little difficult initially, since triangles are in fact different objects but can belong to the same model, so I added in a “parent” attribute to the Triangle class so I could compare the parents of triangles to determine if they were different. As a bonus, the new rays that are for shading can have the shaded colors averaged together to help with aliasing. This feature has not been completely tested and/or debugged but, if desired, should be fairly easy to finish.

## Cel-Shading

Cel-shading is accomplished by simply discretizing the possible range of intensities that the shading function is allowed to output. Basically, the lighting calculations occur for a point as normal, but before returning, the intensity, or magnitude of the color vector, is checked against some intensity “buckets.” If the color is greater than the lower bounds of that bucket, the shaded color is set to the color of the object times the intensity of the upper bounds of that bucket. For my project, I used bucket values of 5-30%, 20-50%, 40-80%, and 100-100%. As the reader can tell, my buckets’ bounds overlap, which is a result of modifying the actual output intensities to make the output images appear brighter. These buckets weights can be further changed to obtain more cartoony looks. The bucket ranges used in the Burnett and Piazza [1] paper were originally used but gave appearances that were too dark. These weights may be revisited in the future.

# Example Images from Finished Project (before presentation)

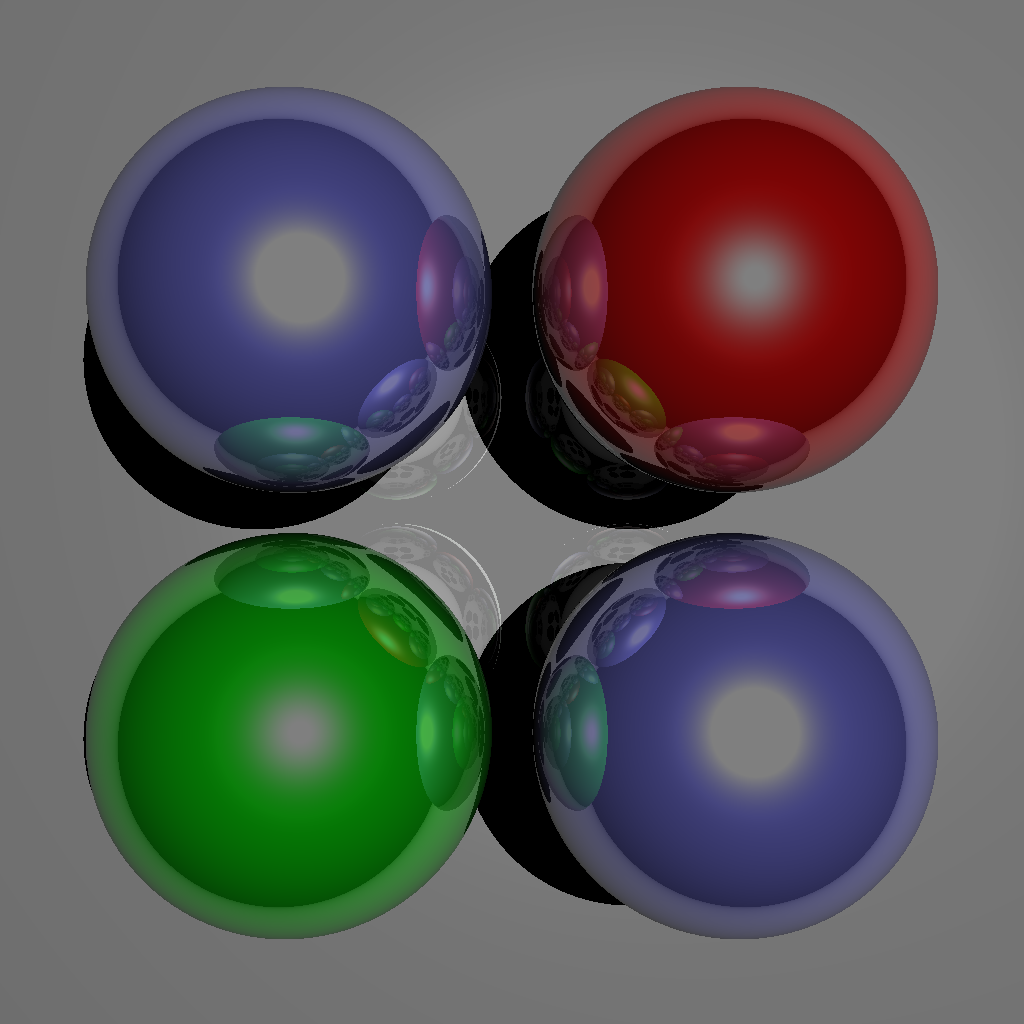


Figure 3. Spheres with no edge silhouettes or cel-shading

A picture containing sport

Description automatically generated

Figure 4. Model of link (from [7]) (no edge silhouettes, cel-shading, or reflections)

A close up of text on a black surface

Description automatically generated

Figure 5. Spheres with edge silhouettes and cel-shading

# Problems Fixed Since Presentation (December 6th)

In my Final Project Presentation, I showed the following image at the very end of my presentation:

A screenshot of a cell phone

Description automatically generated

Figure 6. Edge silhouetted link (attempt)

This image indicates that there where issues in my program associated with triangles, as the spheres in the previous pictures always worked. After hours of debugging, it turns out that the issues stem from the fact that my coordinate system is based on the positive z direction pointing **away** from the camera, not **towards** as is convention. This makes my models (even the cube model taken from the course) have a clockwise orientation in my coordinate system, despite the fact that I use the counterclockwise convention when calculating normals of triangles. This meant that my program was throwing the closest triangle away in favor of the background color due to the reflection checking I did in the raytracing code to make sure it goes through triangles not facing the camera. (Note: it is not implemented correctly at the moment, but it helps find issues with other parts of the program, so I decided to leave it in for now.)

To fix this, I initially just negated the normal calculations wherever they were updated based on the vertex positions (normal = -1 \* calc\_norma() ), but I recently realized I could change the calc\_normal() function to treat the triangle as counter clockwise by switching the order of the cross product. This works for the input cube, but the link model still requires the normal to be inverted, so I may have misunderstood the wavefront obj format that the model came from. In any case, the final image of the project can be seen below:

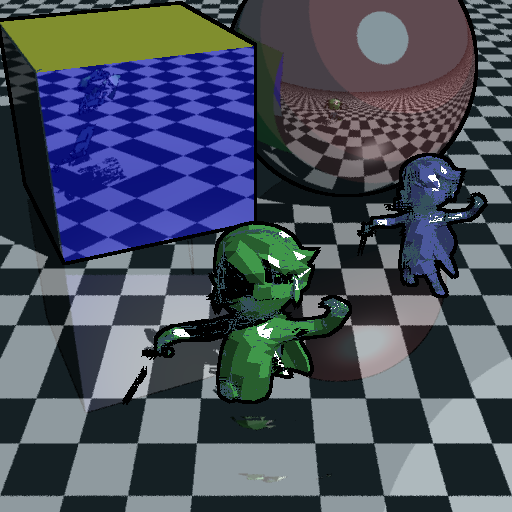


Figure 7. Final Example Output of Raytracing Program (512x512, 1422s = 23min42s)

One can readily see that the reflections of the link models do not appear to be rendered correctly. I would like to say that this is due to some known reason, but at the time of writing, the true cause is up for debate. I speculate that it may be another issue brought about by the inverted coordinate system or a bug in the intersection and reflection parts of the algorithm. In any case, further investigation must be done before this issue is fixed, as over 10 hours of debugging has rendered no solution to this issue. Also, the normals are currently not interpolated over the triangle face. This is due to the format of the model I used for the project. (I discuss this in the “Future Considerations” section.)

A smaller version of Figure 7 was rendered twice, once using the Bounding Box structure and once without. The speed up can be seen in comparing the runtimes of the two pictures (Figure 9), as well as the pictures themselves (Figure 8). The estimated speed increase is about 356 times.

Figure 8. Rendering same image but smaller (16x16) with (right, <1s) and without (left, 318s) the bounding box structure

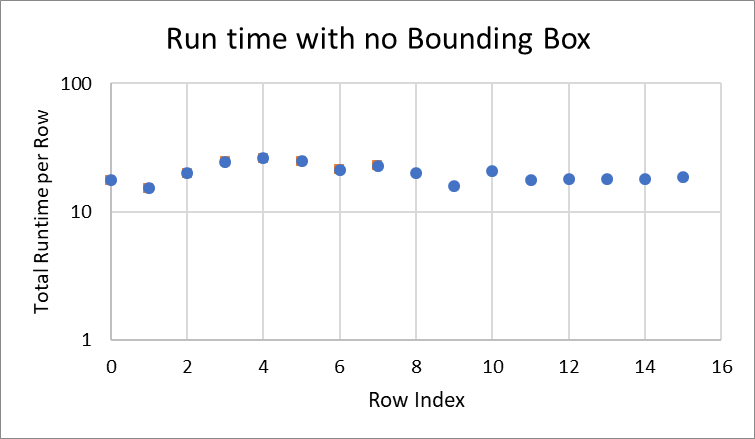
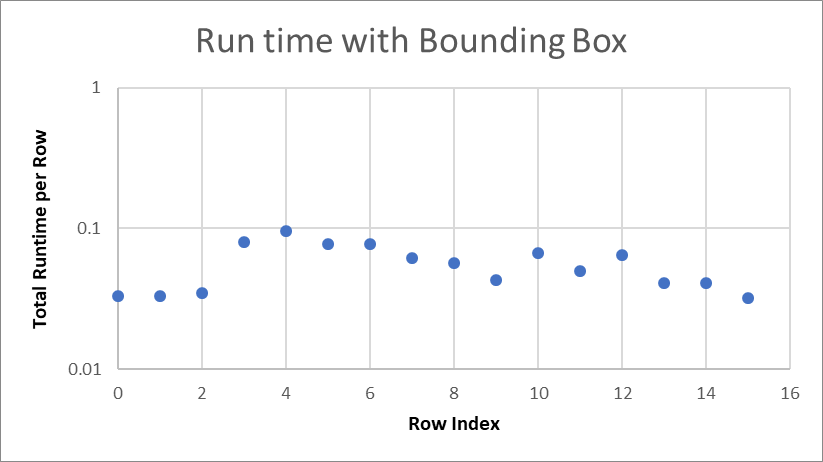
 

Figure 9. Runtimes for the 16x16 images rendered with (left) and without (right) bounding boxes (Note the difference in scale)

I would like to note that the program does, in fact, render the reflections of triangles accurately when not using the added Bounding Box structure, as can be seen below in Figure 10.



Figure 10. Re-render of Figure 7, but without Bounding Box and much smaller (64x64 s 3121s = 52min1s) (arrows added for emphasis on reflections)

# Documentation of Project Code

In order of use/implementation:

* vector.py
  + Class: Vec3
    - Attributes: x, y, z
    - Methods:
      * \_\_init\_\_(x, y, z) : initializes vector with x, y, z members (can use lists, numpy arrays, tuples, and other Vec3s as inputs to x; this would return a Vec3 version or copy of the original)
      * \_\_add\_\_ : element-wise addition
      * \_\_sub\_\_ : element-wise subtraction
      * \_\_mul\_\_ : scalar multiplication
      * dot(other\_vec) : dot product
      * cross(other\_vec) :cross product
      * \_\_abs\_\_ : returns length/magnitude
      * normalize() : returns new vector with same direction but with magnitude of 1
* SceneObjects.py
  + Class: Triangle
    - Attributes: A, B, C (Vec3s), reflectiveness, shininess, parent
    - Methods:
      * \_\_init\_\_(pt\_a, pt\_b, pt\_c, normal, color, reflectiveness, shininess, parent)
      * get\_color(): bit of distancing from attributes to allow CheckeredSphere to work right
      * calc\_normal(): uses points and cross product to calculate the triangle’s normal
      * get\_normal(): bit of distancing to allow spheres to work
      * intersect(ray): determines whether or not ray hits the triangle
  + Class: Sphere
    - Attributes: pos, radius, color, reflectiveness, shininess, parent
    - Methods:
      * \_\_init\_\_(pos, radius, color, reflectiveness, shininess, parent)
      * get\_color(): returns sphere’s color
      * get\_normal(point): returns normal at the point on the sphere
      * intersect(ray): determines whether or not ray hits the sphere
  + Class: CheckeredSphere
    - Methods:
      * get\_color(point): calculates the “on”- or “off”-ness of the sphere at the point
  + Class: PointLight
    - Attributes: position, color, intensity
    - Methods:
      * \_\_init\_\_(position, color, intensity)
* ray.py
  + Class: Ray
    - Attributes: origin, direction, nearest\_hit\_distance, initial\_offset
    - Methods:
      * \_\_init\_\_(ray\_origin, ray\_direction, nearest\_hit\_distance=1e20)
      * reflect(point, normal) : returns a new ray reflected about the given point and using the given normal
  + Class: ShadowRay(Ray)
    - Method:
      * \_\_init\_\_(start\_pos, light) : direction is not normalized in this version
  + Functions:
    - ray\_trace(10 different inputs) : ultimately returns a color to save to a pixel in the main file; uses ray\_intersection() and shade() functions
    - ray\_intersection(ray, objects): loops over objects and returns reference to the one last hit by the ray (closest one)
    - shade(8 inputs): uses the Phong shading model to compute the color of a given point (before reflective specular interactions); also takes into account shadows (only one light currently tested)
* cube.py
  + Functions:
    - cube\_load(reflectiveness): uses the cube\_regl.js as a reference to instantiate a cube; returns a list of triangles representing the cube
* geometry\_loading.py
  + Functions:
    - test\_spheres(size, viewing\_angle): returns list of 4 spheres for use in main.py
    - spheres\_for\_link (size, viewing\_angle): returns list of 1 sphere for use in main.py (typically for demonstrating link model)
    - checkered\_sph\_only(size, reflectiveness, viewing\_angle): returns a list of the single giant sphere used as a floor
    - transform\_objects(list\_of\_objects, transform\_matrix): returns the transformed version of a list of triangles
  + Class: ObjLoader
    - Used to read wavefront obj files. Taken directly from: <https://github.com/totex/PyOpenGL_tutorials/blob/master/ObjLoader.py> Minor changes made to make compatible with the specific model I had.
  + Function: load\_toon\_link(): uses ObjLoader to create and return a list of Triangles representing the Toon Link Model
* BHV\_BBox.py
  + Class: BoundingBox
    - Attributes: left\_box, right\_box, object\_contained, min\_point, max\_point, object\_count
    - Methods:
      * \_\_init\_\_(objects\_list, axis): recursively defines a hierarchical system of bounding boxes surrounding the individual triangles
      * intersect(ray): recursively checks for intersection with contained bounding boxes and the objects contained at the lowest levels; returns a reference to the object hit or False. Based on <https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection>
      * get\_depth(): uses DFS\_count to recursively find how deep the entire structure is.
    - Function: DFS\_count(box): returns the maximum depth of the two sub boxes within the given box + 1, or returns 1 if box contains a triangle\
* main.py
  + Driver file
  + Defines scene setup parameters, size of image, depth of raytrace recursion, and other related parameters
  + Makes use of geometery\_loading.py functions to set up the scene with objects
  + Uses bounding boxes to increase speed of intersection testing
  + Loops over each pixel in image to be generated, using the ray\_trace() function from ray.py to fill in the color
  + Shows and saves the image produced (if desired)
* All extraneous imports used:
  + PIL
  + numpy
  + datetime
  + time
  + transformations
  + os
  + math
  + copy
  + random
  + Numbers
  + builtins

# Noteworthy\* Data from Project

Total “Lines of Code” in final code base: ~915

Total “Commented-Out”\*\* Lines: ~138

Number of Triangles in Link model used: 5,860

Speed Increase from Using Bounding Boxes: ~356x faster (at least in 16x16 case)

\*In my opinion

\*\*For various reasons, debugging print statements, temporary fixes, defunct uses, etc.

# Personal Takeaways

1. Even if the reason behind the choice for the “camera faces -z direction” convention is not understood, it should still be used, since any model downloaded will (most likely) have that included in them, and graphics programmers are (almost) naturally conditioned to define any model used with that convention in mind. This can lead to the issues I faced when using a reversed coordinate system.
2. The bounding box/hierarchical volume structure should have been implemented much sooner in development, as it greatly increased speed. While this was expected, I did not expect the code overhead to be so small. It is my understanding that premature optimization is almost always a bad choice, but now it is evident that if an optimization been confirmed to be help multiple times, it may be good to implement.
3. Despite starting several weeks ahead of some of the other students, I ran into difficulties with the final touches to the project in the few days before the deadline. Significant progress was made, but still not everything desired was accomplished. I should have kept a document stating the exact goals of the project and the current progress on each of those goals, as some time may have been wasted on features that were not actually meant to be in the project anyway (such as trying to implement normal interpolation).

# Future Considerations for this Project

Fixing the reflections will be priority #1 if should I decide to improve upon this project. I am unsure how long that would take, and I fear that I would need to completely rewrite large portions of the code base to make it work.

Another aspect of this code I would hope to improve would be the interpolation of the normals around a triangle in the shading function. The model of Link that I used had each vertex only storing the normal for the entire triangle, so interpolation would be useless for this particular model. However, different model formats contain the normals for each individual vertex, making interpolation useful again. I hope to find such a model soon and see if I could get that smoothing working. Alternatively, I could include some extra interpreter function to save the average of the normals of the triangles that a vertex is a part of as the normal for that vertex, and interpolate these averaged normals, but I do feel that this would cause some sort of loss of detail or extra rounding of edges.

Finally, I would like to actually incorporate the use of numpy arrays to make the raytracing work more quickly. There is a method used in [2] that I attempted to use, but the syntax was not clear to me. After more devotion to understanding such use, I may be able to make this ray tracer function even faster. This project may never reach “real-time” capabilities, but it would still be neat to try.

One last picture for the road. With everyone this time:

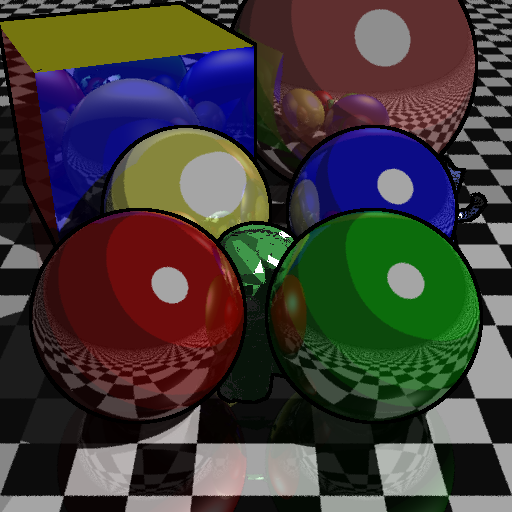


Figure 11. Rendering Link with the cube and spheres (last image)

# References

|  |  |
| --- | --- |
| [1] | A. Burnett and T. Piazza, "Toon Shading Using Distributed Raytracing," Rensselaer Polytechnic Institute, Troy, NY, 2017. |
| [2] | "A reasonably speedy Python ray-tracer," excamera, [Online]. Available: https://excamera.com/sphinx/article-ray.html. [Accessed December 2019]. |
| [3] | P. Shirley and S. Marschner, Fundamentals of Computer Graphics, A K Peters, Ltd, 2009. |
| [4] | Henrik, "Ray\_trace\_diagram.svg," Wikipedia, 2019. |
| [5] | Scratchapixel, "A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.)," Scratchapixel, [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection. [Accessed December 2019]. |
| [6] | A. Williams, S. Barrus and R. K. M. &. P. Shirley, "An Efficient and Robust Ray-Box Intersection Algorithm," *Journal of Graphics Tools,* vol. 10, no. 1, pp. 49-54, 2005. |
| [7] | ThatOneGuyWhoDoesThings, "[SSBB] Toon Link," SketchFab, 2015. [Online]. Available: https://sketchfab.com/3d-models/ssbb-toon-link-c4737326bee4445ab2a565952ad32eab. [Accessed November 2019]. |