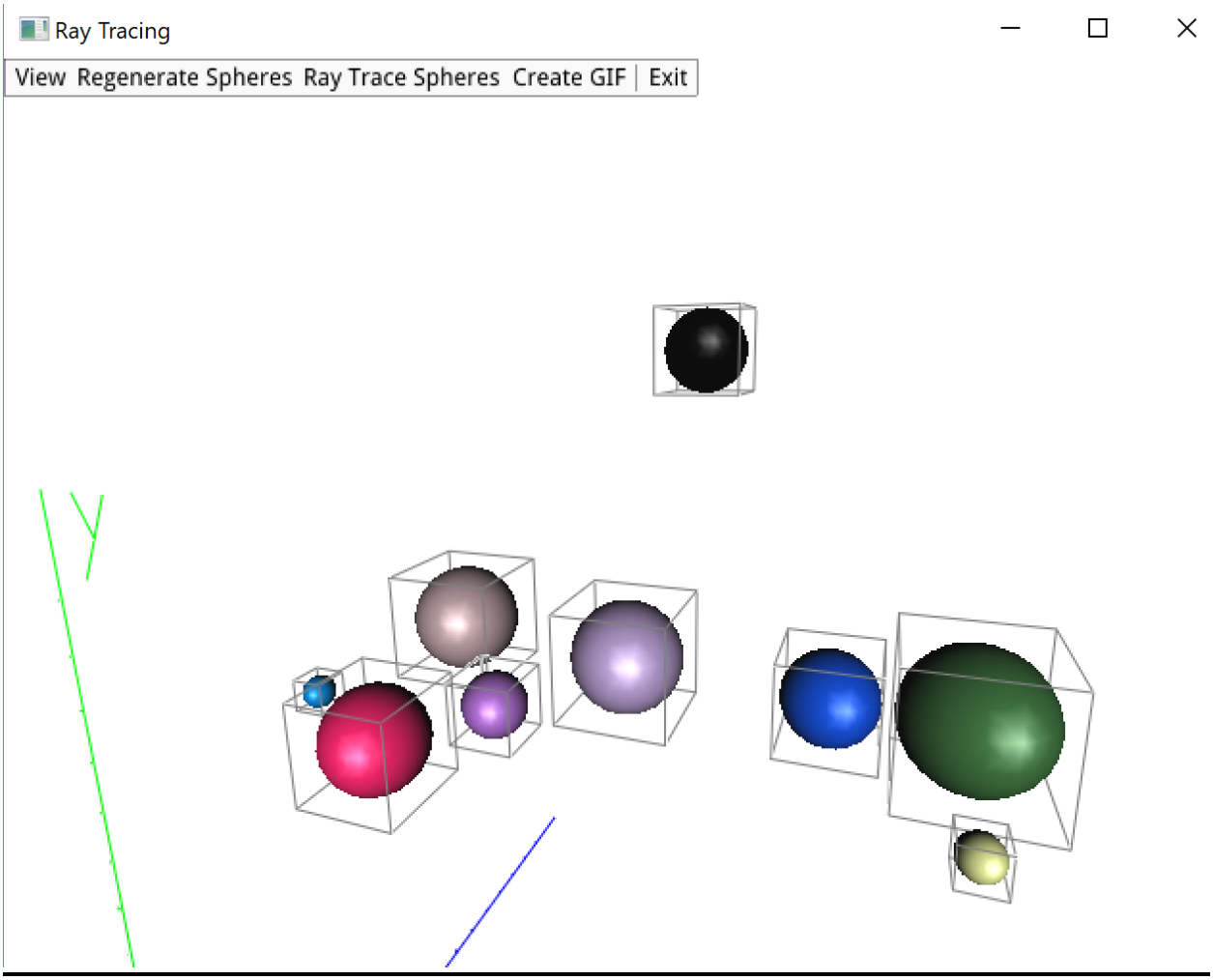
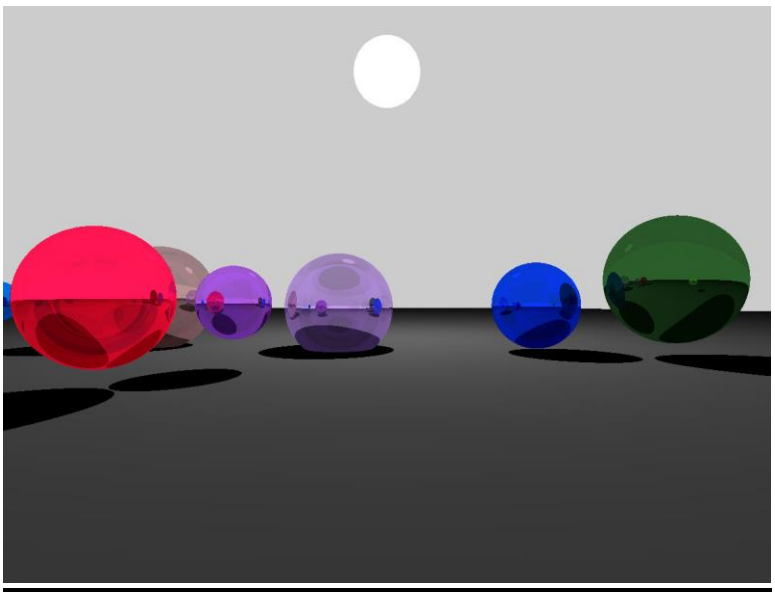
EECS 267 Final Project – Ray Tracing





Main Points

Sphere Class

Holds general information about the sphere needed for ray tracing. Also contains a function that checks for ray intersections. (Source: Scrathapixel)

```
GsVec center;
float radius;
float radiusSquared;
GsVec surfaceColor;
GsVec emissionColor;
float transparency;
float reflection;
```

Randomize Sphere Scene

The code can be found under the function:

```
void MyViewer::build_scene ()
```

This function basically randomizes the location, color, and size of the spheres (using rand() and time as a seed for the srand() function).

The light source is statically set.

A huge sphere is also statically set underneath all of the generated spheres (to create a surface to project the shadows on).

The randomly generated spheres can be placed anywhere from:

X Value: [-6, 6].

Y Value: [0, 1].

Z Value: [-15, -35].

The light source is centered at (0, 20, -40).

(Our viewpoint of the ray traced scene is from the origin, facing towards the negative z-axis)

The generated spheres take on random colors.

The generated spheres can have a radius from 1 to 4.

See function for more details.

Ray Trace Renderer

The code can be found under the function:

```
void rayTraceRender(vector<Sphere>& mySpheres)
```

This function is called when we click on “Ray Trace Spheres”. For each pixel, it shoots out a ray from (0,0,0) and calls traceRay(). See traceRay() below for more details.

After shooting a ray out for every pixel (width*height pixels), we then output each color of each picture to a PPM file. We can open this PPM file in GIMP, or convert it to a JPG.

Tracing Rays

The code can be found under the function:

```
GsVec traceRay(GsVec rayOrigin, GsVec rayDirection, vector<Sphere> mySpheres, int
depth)
```

This function is where all the ray tracing occurs. Here’s a brief overview:

Given the origin of a ray, and its direction, we want to see if it hits any of our spheres.

If it doesn’t hit any spheres, return the background color.

If it does hit a sphere, keep note of that sphere. (Let’s call it hitSphere).

Once we find a hitSphere, we want to check if it’s transparent or reflective.

If it’s transparent, we want to shoot a ray of refraction through the hitSphere. (Equation to calculate the refraction ray direction is from: Scratchapixel). This new ray is then recursively calling traceRay() to see where it intersects.

If it’s reflective, we want to shoot a ray of reflective from the hitSphere. (Equation to calculate the reflective ray direction: Scratchapixel). This new ray is then recursively calling traceRay() to see where it intersects.

This recursive calling of new rays has a depth limit (passed as a parameter into traceRays()). The current depth limit is 5.

If the sphere isn’t transparent or reflective (or if the recursive depth limit has been reached), we want to see if the light source is shining.

Basically, we check if the light source's ray of light is hitting some sphere. If it is, then we change the transmission of that light ray to 0. (Equation to calculate the light ray direction is from: Scratchapixel).

The resulting color of a pixel is a mix of the base surface color of the hitSphere, and the colors of the spheres that were hit by the rays of transparency and reflection. (Mixing equation from: Scratchapixel).

Utility Functions

The code can be found under the functions:

```
GsVec multVecs(GsVec a, GsVec b)
```

Used to multiply the coordinates of two vectors together to get a new vector. (Like a dot product, except we don't add, we just save the multiplied values and make a new vector).

```
bool intersect(GsVec rayOrigin, GsVec rayDirection, float &t0, float &t1)
```

Used to check for ray-sphere intersection

(UI) Events

Here's what each key does in this project:

Regenerate Spheres : Randomly regenerate a new scene of spheres.

Ray Trace Spheres : Perform a ray tracing of the current scene of spheres. It'll be saved as a PPM file in the vs2017 folder. Filename: "RayTraceTest#.ppm" Where # is n^{th} number of ray tracings performed in the current session.

Create GIF : This will create 40 Ray Tracings of the current scene of spheres. Each ray tracing has a unique position of the light source (following an orbital path). (Basically the light source is rotated around the z-axis by an increment before each ray trace). We can convert these 40 PPM files into 40 JPG files. From there, we can make a GIF of 40 JPG files.

Note: This doesn't have to always be 40 ray tracings, we can change the number of ray tracings in the code. As well as the resolution (pixel width and height).

Evaluation

Topic

For this project, I want to evaluate the following:

1. The Runtime of a single ray tracing of the current scene of spheres.
2. The output PPM file size of a single ray tracing of the current scene of spheres.
3. Memory usage during a single ray tracing of the current scene of spheres.

Variables of the evaluation:

1. The number of spheres on the scene.
2. The resolution of the image. (Pixel width and height).
3. Depth of recursive calls to traceRay()

Purpose

The reason why I want to evaluate my project with regards to runtime time, file size, and memory allocation, is because I want to see what the optimal number of spheres on the scene and resolution is.

Statistics/Table

Ran on the following processor and graphics card:

1) *Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 2592 Mhz, 2 Core(s), 4 Logical Processor(s)*
2) *Intel(R) HD Graphics 520 display adapter*

The number of spheres was controlled via parameter in the for loop in build_scene().

The resolution of the PPM image was controlled via “width” and “height” parameters in rayTraceRender()

The depth of recursive calls to traceRay() was controlled via MAX_RAY_DEPTH parameter.

The Creation Time was measured using *gs_time()*

PPM file size was measured simply using the File Explorer on Windows 10.

Memory Allocation was measured using the Diagnostics tool on Visual Studio 2017.

Number of Spheres	Depth of Recursion	Resolution (Width x Height)	Runtime (seconds)	Output PPM file size	Memory Allocation
5	5	640 x 480 px	1.94 seconds	901 KB	44 MB
5	5	1200 x 900 px	10.9 seconds	3,165 KB	52 MB
5	5	3200 x 1800 px	27.2 seconds	16,876 KB	112 MB

5	10	3200 x 1800 px	71.1 seconds	16,876 KB	112 MB
7	5	640 x 480 px	2.61 seconds	901 KB	52 MB
7	5	1200 x 900 px	8.77 seconds	3,165 KB	54 MB
7	10	1200 x 900 px	21.3 seconds	3,165 KB	54 MB
7	5	3200 x 1800 px	38.3 seconds	16,876 KB	113 MB
10	5	640 x 480 px	5.10 seconds	901 KB	43 MB
10	10	640 x 480 px	8.56 seconds	901 KB	45 MB
10	5	1200 x 900 px	15.8 seconds	3,165 KB	54 MB
10	5	3200 x 1800 px	87.7 seconds	16,876 KB	121 MB

Analysis

From the data above, we can see that the runtime has a direct relationship with the resolution & number of spheres & depth of recursion.

As the resolution or number of spheres or depth of recursion increases, so does the runtime time.

Memory allocation increases with the number of spheres in the scene, regardless of resolution or depth of recursion.

Output PPM file size increases as the resolution increases, regardless of the number of spheres in the scene or depth of recursion.

Possible experimental errors:

1) How closely the spheres are randomly placed next to each other (causing more reflection).
2) Left over memory from the previous run.

Conclusion

As usual, the optimal choice for this ray tracing project depends on how we are going to apply this project.

If we want to apply this project in a dynamic environment where the scene changes (such as the light source moving), we would want to use a lower resolution and possibly a lower depth of recursion, so it'll compute the PPM images faster (since it will have less pixels and recursive rays to compute). An optimal choice for this would be selecting a resolution of 1200 x 900 pixels and a depth limit of 5.

If we want to apply this project in a static environment, we won't have to worry so much about runtime, since we are only running it only once. In this case, we can pick a higher resolution to have a smoother display of the static scene. However, this will increase the size of our output PPM file. An optimal choice for this would be selecting a resolution of 3200 x 1800 pixels. I

wouldn’t worry so much about increasing the depth limit of recursion. A depth limit of 5 captures a lot of detail already; increasing it to 10 may not be too noticeable. (Although, having a higher resolution means we can see more detail, so maybe it may be noticeable).