CS 175 - Final Project

Ye Zhao: yezhao@college.harvard.edu

December 15, 2012

1 Introduction

This project implements a simple ray tracer using the approach illustrated in the book, "An Introduction to Ray Tracing" by Andrew S. Glassner. The ray tracer implments the basic geometrical shapes such as sphere, plane, box, cylinder. It also extends the implementation in class to include refraction and super sampling.

2 Simulating Refraction

2.1 Background

When a light ray travels from one medium to another, the speed of light changes which results in a change in the direction of travel of light according to the Snell's law,

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \quad \Rightarrow \quad \sin\theta_2 = \frac{n_1}{n_2}\sin\theta_1 \tag{1}$$

Since the sin of an angle cannot be greater than 1. We will have total internal reflection if

$$\frac{n_1}{n_2}\sin\theta_1 > 1 \quad \text{or} \quad \frac{n_2}{n_1} > \sin\theta_1 \tag{2}$$

which is a condition for which the ray will not be transmitted.
To calculate the direction of the transmitted ray, we using vector calculation to derive the formulas for the stransmitted ray.

$$\vec{t} = \vec{t}_{\parallel} + \vec{t}_{\perp} \tag{3}$$

$$\left|\vec{t}_{\parallel}\right| = \left(\frac{n_1}{n_2}\right)\left|\vec{i}_{\parallel}\right| = \left(\frac{n_1}{n_2}\right)\left(\vec{i} - \cos\theta_i \vec{n}\right) \tag{4}$$

where $\cos\theta_i = \vec{i}\cdot\vec{n}$, $\vec{i}$ is the incident ray, $\vec{t}$ is the transmitted ray and $\vec{n}$ is the normal to the surface. Using the Pythagoras' Theorem, we see that

$$\vec{t}_{\perp} = \sqrt{1 - \left|\vec{t}_{\parallel}\right|^2}\vec{n} \tag{5}$$

Sustitute this back into the equation for the transmitted ray, we get

$$\vec{t} = \vec{t}_{\parallel} + \vec{t}_{\perp} \tag{6}$$

$$\vec{t} = \left(\frac{n_1}{n_2}\right)\vec{i} - \left[\left(\frac{n_1}{n_2}\right)\cos\theta_i + \sqrt{1 - \left|\vec{t}_{\parallel}\right|^2}\right]\vec{n} \tag{7}$$

$$\vec{t} = \left(\frac{n_1}{n_2}\right)\vec{i} - \left[\left(\frac{n_1}{n_2}\right)\cos\theta_i + \sqrt{1 - \sin^2\theta_t}\right]\vec{n} \tag{8}$$

$$\tag{9}$$

This gives me three equations that I can then use to find the refracted ray, ie

$$\cos\theta_i = \vec{i} \cdot \vec{n} \tag{10}$$

$$\sin^2\theta_t = \left(\frac{n_1}{n_2}\right)^2 \left(1 - \cos^2\theta_i\right) \tag{11}$$

$$\vec{t} = \frac{n_1}{n_2}\vec{i} - \left(\frac{n_1}{n_2}\cos\theta_i + \sqrt{1 - \sin 2\theta_t}\right)\vec{n} \tag{12}$$

2.2 Code

The code for refraction is given below

```
// Refraction
int refraction (Ray &ray, Color &color, Object *object, Scene &scene,
    Cvec3f &N, Cvec3f &pointOfIntersection, int depth, double &reflectionConstant,
    double refractionIndex0, double &distance, int &hit) {
    double refractionIndex = object->getMaterial()->getRefractionIndex();
    double n = refractionIndex0 / refractionIndex;
    Cvec3f N1 = N * hit;
    double cosThetaI = - dot(N1, ray.getDirection());
    double sinThetaI = sqrt(1.0 - cosThetaI * cosThetaI);
    double sinThetaT = n * sinThetaI;

    if (sinThetaT * sinThetaT < 1.0) {
        double cosThetaT = sqrt(1.0 - sinThetaT * sinThetaT);
        Cvec3f R4 = ray.getDirection() * n - N1 * (n * cosThetaI + cosThetaT);
        R4.normalize();
        Color refractedColor(0.0, 0.0, 0.0);
        double dist;
        Cvec3f R5 = pointOfIntersection + R4 * EPSILON;
        Ray R6 = Ray(R5, R4);
        rayTrace(R6, refractedColor, scene, depth + 1, refractionIndex, dist);
        // Beer's Law
        Color absorbance = object->getMaterial()->getColor() * 0.15 * dist * (-1.0);
        Color transparency = Color(exp(absorbance[0]),
            exp(absorbance[1]), exp(absorbance[2]));
        color += refractedColor * transparency;
    }
    return 0;
}
```

2.3 Resulting Image

The resulting image is given in Figure 1. As you can see in the figure, the light grey sphere at the front has a combination of reflection and refraction with no specular or diffuse reflection. There are three same sized spheres at the back colored, red, yellow and green. As expected, these colored spheres are inverted in the bigger transparent sphere in the front. Also since reflection is enabled, there are multiple images of the colored spheres due to the multiple reflections off the front and back of the sphere.

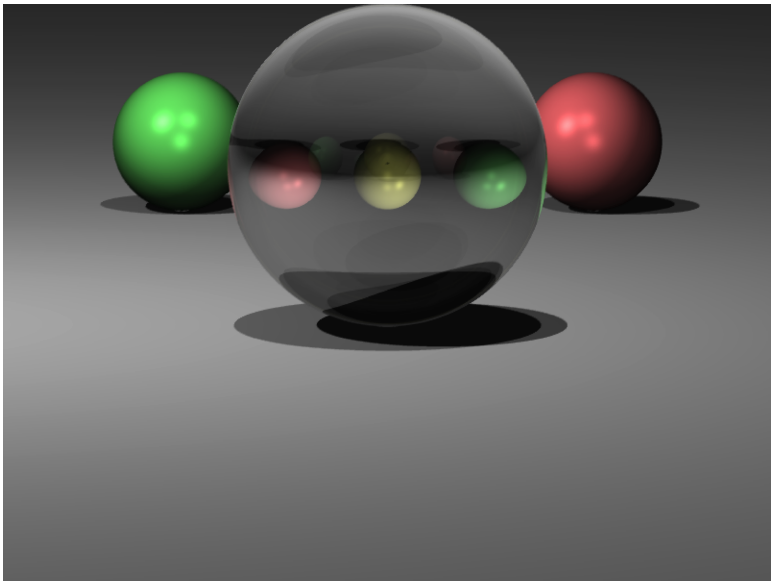

Figure 1: This figure demostrate the effect of implementing refraction.

3 Supersampling

3.1 Background

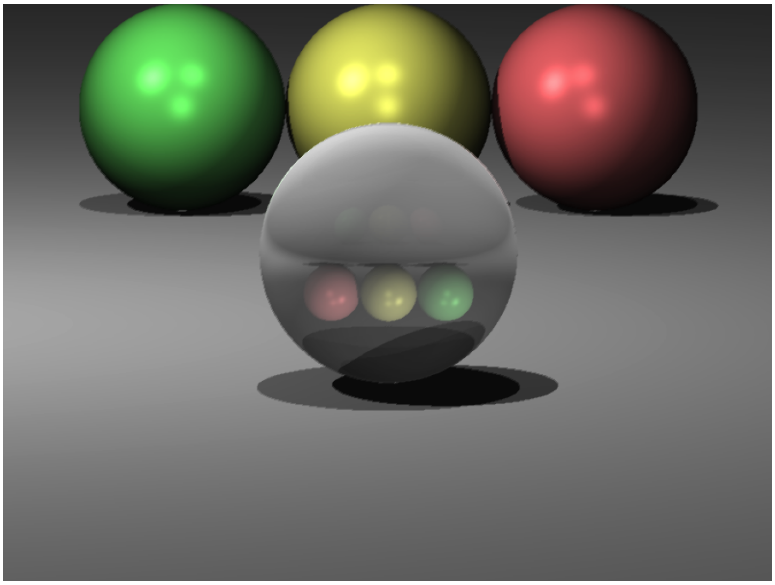

Figure 2: This image has jagged edges due to the aliasing of the pixels.

The ray tracing algorithm that I implemented was only sending rays into a scene in a grid pattern that matches the image pattern. This produces images with jagged edges as seen in Figure **??**. The jagged edge can be most easily seen along the boundaries of the shadows.
To smooth out the edge, I implemented an antialiasing technique using a grid supersampling technical. In supersampling, more rays are sent into the scene. Each of these additional rays are slghly offset to the rays that were used initially. The color of the pixel is then the sum of the color of each

ray averaged over the total number of rays sent. The supersampling technique is easy to implement but more computationally intesnsive.

3.2 Supersampling Code

The following code use a simple grid technique for supersampling.

```
int s = 5;
double invS = 1.0 / (double) (s - 1);
.
.
.
    for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; i++) {
            color = Cvec3f(0.0, 0.0, 0.0);
            for (int dy = -s/2; dy < (s+1)/2; dy++) {
                for (int dx = -s/2; dx < (s+1)/2; dx++) {
                    double x = lookAt[0] + dx * invS / 100. + i / 100.;
                    double y = lookAt[1] + dx * invS / 100. + j / 100.;
                    double z = lookAt[2];
                    double R = sqrt(x * x + y * y + z * z);
                    double gamma = 1 / sqrt(1 - beta * beta);
                    double z_transform = (lookAt[2] + beta * R) * gamma;
                    direction = Cvec3f(x, y, z_transform) - viewPoint;
                    direction.normalize();
                    ray = Ray(viewPoint, direction);
                    object = rayTrace(ray, color, scene[0], depth, refractionIndex, distance);
                }
            }
            int b = color[0] * 255 / (s * s);
            int g = color[1] * 255 / (s * s);
            int r = color[2] * 255 / (s * s);
            int index = j * width + i;
            if (r > 255) r = 255;
            if (b > 255) b = 255;
            if (g > 255) g = 255;
            image_data[index * 3 + 0] = (unsigned char) b;
            image_data[index * 3 + 1] = (unsigned char) g;
            image_data[index * 3 + 2] = (unsigned char) r;
```

3.3 Supersampling Image

4 Box Ray Intersection

4.1 Determining the Intersection

Another elementary geometric shape is a box. A method for performing the intersection between a ray and a box was developed by Kay and Kajiya. This method uses slabs between two parallel planes. The box is axis aligned such that the first slab is between the x_{min} and x_{max}. The second

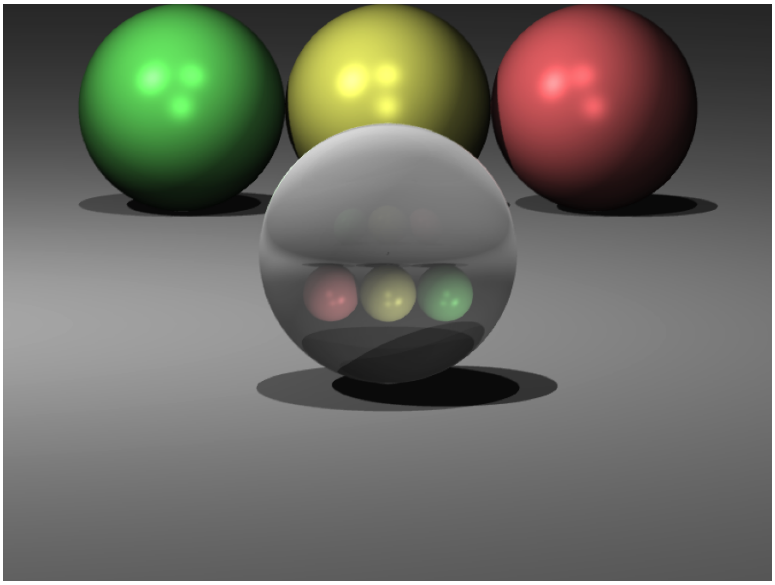

Figure 3: This image is rendered using the grid supersampling technique at a supersampling ratio of 5. The resulting image has a much smoother edge as compared to the image without supersampling.

slab is between y_{min} and y_{max}. And the third slab is between z_{min} and z_{max}. The intersection of these three slabs determine the box.
In this implementation, there are two points that defines the box: the first point at the minimum x, y and z location; and the second point at the maximum x y, z location. The algorithm starts off by initializing the distance to the point of intersection to a large negative numeber and the point of exit of the ray to a large positive number. Then a ray is shot towards the box to determine its intersections with the x, y and z slabs.
To determine if the ray is coming from the minimum or the maximum direction, I start my rays coming from some minimum direction, such as $-z$, and then place all my objets in the $+z$ direction. Then I check my near and far values. The largest near value and the smallest far value is needed to determine the point of intersection. If the near value is greater than the far value, the box is missed. If the far value is less than zero, then the box is in the posite direction from the ray which means that the ray misses the box and return a miss value. If the near value is less than zero but the distance to the far value is still psotive, then the ray is in the nox. A value indicating that we are in the box is then returned. Otherwise the distance is modified to be the near value and a hit is returned.

4.1.1 Box Ray Intersection Code

The code that implements the box ray intersection is given below:

```
int Box::intersect(Ray &ray, double &root) {
    double tmp, tnear=-1.0e6, tfar=1.0e6;
    Cvec3f tmin = (minPoint - ray.getOrigin()) / ray.getDirection();
    Cvec3f tmax = (maxPoint - ray.getOrigin()) / ray.getDirection();
    if (tmin[0] > tmax[0]) {
        tmp=tmin[0];
        tmin[0]=tmax[0];
```

```
        tmax[0]=tmp;
    }
    if (tmin[1] > tmax[1]) {
        tmp=tmin[1];
        tmin[1]=tmax[1];
        tmax[1]=tmp;
    }
    if (tmin[2] > tmax[2]) {
        tmp=tmin[2];
        tmin[2]=tmax[2];
        tmax[2]=tmp;
    }
    tnear = max(tmin[2], max(tmin[1], max(tmin[0], (float) tnear)));
    tfar = min(tmax[2], min(tmax[1], min(tmax[0], (float) tfar)));

    if (tnear > tfar) return 0;
    if (tfar < 0) return 0;
    if (tnear<0) { root=tfar; return -1; }
    root = tnear;
    return 1;
}
```

4.2 Determining the Normal

The normal is much easier to determine since the box is axis aligned. The normal for the six sides of the box are also known a priori The point of intersection lies on one of the six faces. Therefore, subtracting the point of intersection from the minimum value for the corner of the box and then doing the same with the maximum value for the corner of the box shoul find the correct face.

4.2.1 Box Normal Code

The code for determining the normal of the box is given below:

```
Cvec3f Box::getNormal(Cvec3f &intersectionPoint) {
    Cvec3f distance = minPoint - intersectionPoint;
    double minDistance = abs(distance[0]);
    int side = 0;
    if(abs(distance[1]) < minDistance) {
        minDistance = abs(distance[1]);
        side = 2;
    }
    if (abs(distance[2]) < minDistance) {
        minDistance = abs(distance[2]);
        side = 4;
    }
    distance = maxPoint - intersectionPoint;
    if (abs(distance[0]) < minDistance) {
        minDistance = abs(distance[0]);
        side = 1;
```

```
    }
    if (abs(distance[1]) < minDistance) {
        minDistance = abs(distance[1]);
        side = 3;
    }
    if (abs(distance[2]) < minDistance) {
        minDistance = abs(distance[2]);
        side = 5;
    }
    if (side == 0) return Cvec3f(-1., 0., 0.);
    if (side == 1) return Cvec3f( 1., 0., 0.);
    if (side == 2) return Cvec3f( 0.,-1., 0.);
    if (side == 3) return Cvec3f( 0., 1., 0.);
    if (side == 4) return Cvec3f( 0., 0., -1.);
    return Cvec3f(0., 0., 1.);
}
```

4.3 Resulting Image

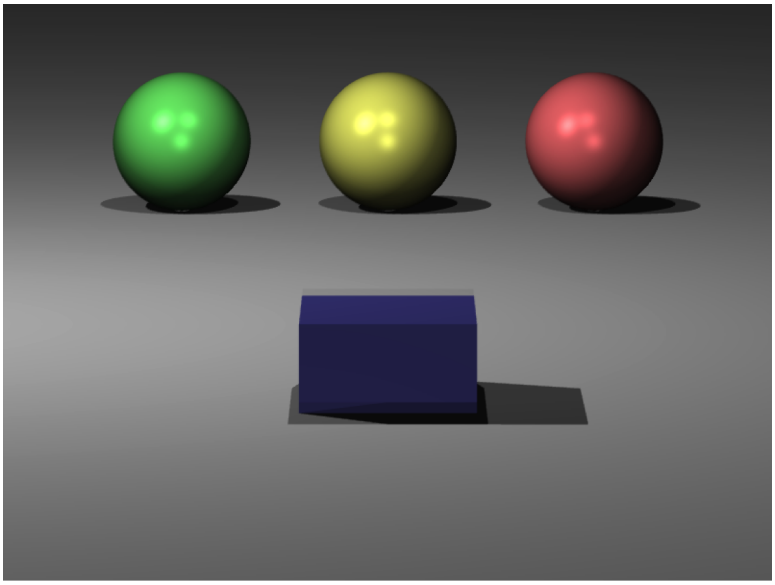

Figure 4: This shows an opaque box in front of the three opaque spheres.

The resulting image for the box algorithm can be seen in Figure 4. This is an opaque box shaded in blue. The same algorithm can also be used to generate a translaucent box that shows the refracted rays passing through it as in Figure 5.

5 Ray Cylinder Intersection

5.1 Background

A cylinder is ususally defined as haiving a center, a height, a known radius and some known orientation. In my case, I am restricting the cylinder to be alighed to the y axis, that it is a circular

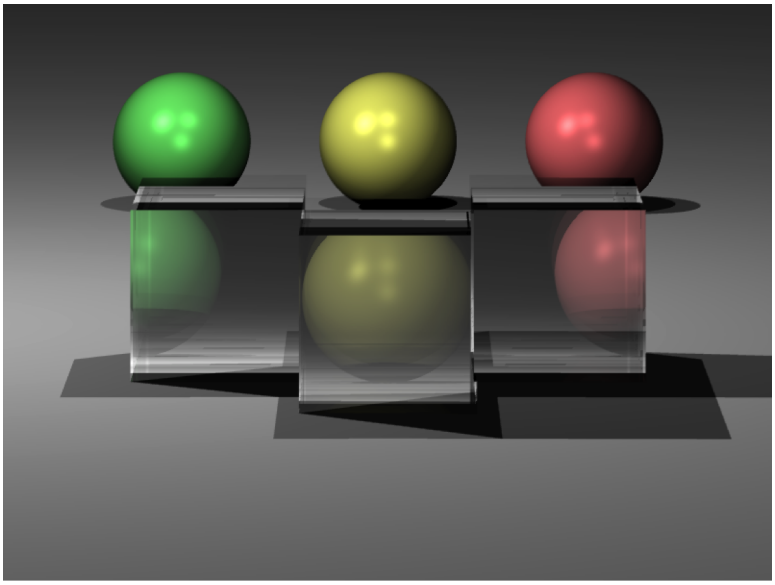

Figure 5: This shows three transparent box in front of the three opaque spheres.

cylinder and that it is not an infitely long cylinder, such that

$$(x - x_c)^2 + (z - z_c)^2 = R^2 \quad y_{min} \leq t \leq y_{max} \tag{13}$$

where x and z lie on the surface of the cylinder, x_c and z_c denotes the center of the cylinder an R^2 is the radius squared.

5.2 Cylinder Intersection

Two things can happen between a ray and a cylinder: totally missing the cylinder or intersecting the cylinder. If a ray intersects the cylinder, it can either start outside the cylinder and intersect it twice, or intersect it once at the tangent , or start inside the cylinder and intersect it once.
To find the intersection between a ray and a cylinder, we substitute the parametric ray equations

$$x = x_0 + x_d t \qquad\qquad z = z_0 + z_d t \tag{14}$$

into the cylinder equation as

$$(x_0 + x_d t - x_c)^2 + (z_0 + z_d t - z_c)^2 - R^2 = 0 \tag{15}$$

$$(x_o + x_d t - x_c)^2 = (x_0 - x_c)^2 + 2x_d(x_0 - x_c)t + (x_d^2)t^2 \tag{16}$$

This is simply the quadratice equation in t, where

$$A = x_d^2 + z_d^2 \tag{17}$$

$$B = 2(x_d(x_0 - x_c) + z_d(z_0 - z_c)) \tag{18}$$

$$C = (x_0 - x_c)^2 + (z_0 - z_c)^2 \tag{19}$$

This solves the quadratic equation

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{20}$$

If there is no solution to the equation, then the ray doesn't intersect with the cylinder. If the determinant is less than zero, then the equation has no root. If the determinant is equal to zero, then we are taking the square root of zero. Then we have two repeated roots. The ray hit the cylinder. If the determinant is greater than zero, we have two repeated roots. If one root is negative and one root is positive, then the ray started inside the cylinder. Extending the ray forward gives a positive root which is the intersection of the cylinder. If both roots are positive, then the ray started outside the cylinder and intersected the cylinder twice. The first interstection is where the ray enters the cylinder.

5.2.1 Cylinder Intersection Code

The code for the intersection of the cylinder is given below:

```
int Cylinder::intersect(Ray &ray, double &root) {
    Cvec3f Rd = ray.getDirection();
    Cvec3f Ro = center - ray.getOrigin();
    Cvec3f intersectionPoint;
    double a = Rd[0]*Rd[0] + Rd[2]*Rd[2];
    double b = Ro[0]*Rd[0] + Ro[2]*Rd[2];
    double c = Ro[0]*Ro[0] + Ro[2]*Ro[2] - radiusSquare;
    double disc = b*b - a*c;
    double oldRoot = root;
    double d, root1, root2;

    int returnValue = 0;

    if (disc > 0.0) {
        d = sqrt(disc);
        root1 = (b - d) / a;
        root2 = (b + d) / a;

    if (root2 > 0) {
        if(root1 < 0) {
            if(root2 < root) {
                root = root2;
                returnValue = -1;
            }
        }
        else {
            if(root1 < root) {
                root = root1;
                returnValue = 1;
            }
        }
    }
    return returnValue;
}
```

5.3 Cylinder Normal

Since the cylinder is aligned on the y axis, the returned normal would be

$$(P_x, 0, P_y) \tag{21}$$

where P is the point of intersection. However, at an arbitrary location, we need to check to see where the point of intersection lies in relation to the center of the cylinder. The code looks like this:

```
Cvec3f Cylinder::getNormal(Cvec3f &intersectionPoint) {
    if(intersectionPoint[1] == top[1]) {
        return Cvec3f(0., 1., 0.);
    }
    if (intersectionPoint[1] == bottom[1]) {
    return Cvec3f(0., -1., 0.);
    }
    Cvec3f normal = (intersectionPoint - center) / radius;
    normal[1] = 0.0;
    normal.normalize();
    return normal;
}
```

5.4 Resulted Image

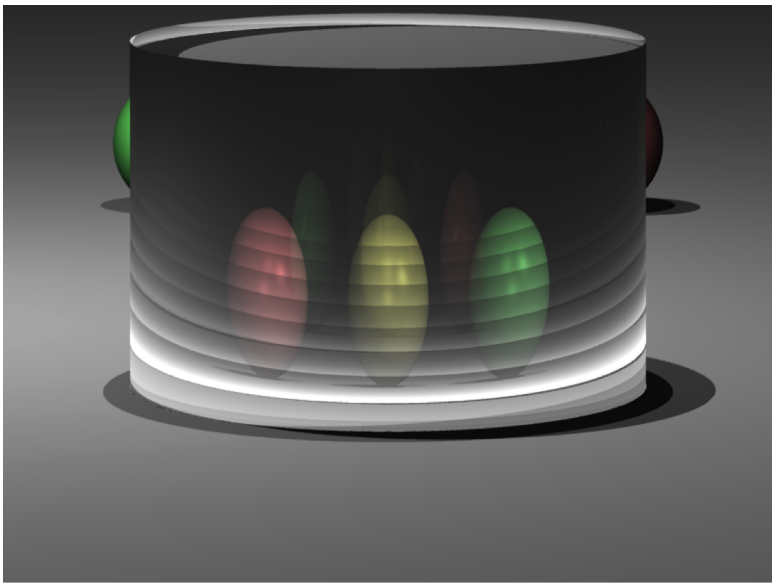

Figure 6: This shows refracted images of the three spheres coming through the cylinder.

The resulte image from the implementation of the cylinder algorithm is captured in Figure 6.

6 Special Relativity

Raytracing is good for visualizing special relativity effects since the geometry can be controlled by the transformation of the coordinates.

6.1 Theorey and Implementation

Consider two coordinate systems which we refer to as the primed and unprimed coordinate systems. The origins of both systems coincide at time zero, as measured by observers placed at the origin of each system. The unprimed system has a velocity v along the z axis of primed system. An observer travelling with the unprimed system records an event at time t and location (x, y, z) which we indicated by the four-vector (x, y, z, t). Special relativity tells us that an observer travelling with the primed system will record the event (x', y', z', t') where the relationship between the primed and unprimed coordinates is given by

$$x' = x \tag{22}$$
$$y' = y \tag{23}$$
$$z' = \gamma(z + \beta t) \tag{24}$$
$$t' = \gamma(t + \beta t) \tag{25}$$

where $\beta = \frac{v}{c}$ and $\gamma = \frac{1}{\sqrt{1-\beta^2}}$.
In the situation of the simple model of relativisitc effect for this program, we assume that the observer is moving relative to the stationary objects on the ground. Hence to determine the relativistic ray intersection, we first transform the ray from the frame of the observer to the frame of the ground. We then shoot out the ray to determine the intersection between the ray and the the objects on the ground.

6.2 Resulted Images

The following shows a series of images for different values of β rendered by our equation. The image where the camera is at rest with ground shows an array of 12 spheres. When the speed of camera is increased to near the speed of light, we see distinct distortion of the shape of these spheres due to Lorentz transformation.

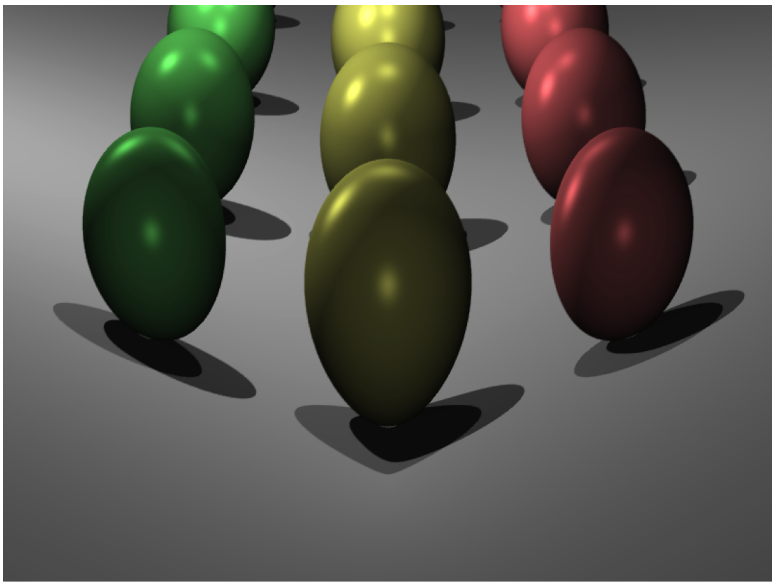

Figure 7: $\beta = -0.9$.

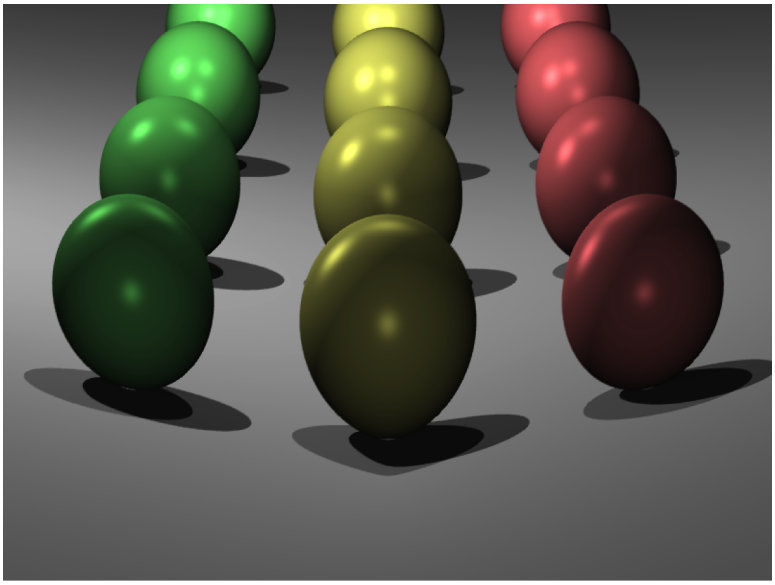

Figure 8: $\beta = -0.75$.

7 Conclusions

This project implements the various primatives for the raytracing program. Several problems were encountered when coding up the ray tracing algorithm. First, there are ghost dots appearing when the rebounced ray is not set far away enough from the surface of intersection. Second, due to supersampling, the time taken to render the image increased by a significant amount and therefore we have to figure out a compromise solution.
For the future, it will be good to implement the ray tracing on a GPU as this will increase the run time by a significant amount. Also, for the special relativity section, due to the time constraint, only lorentz transformation was simulated. It will visually more realistic to also include other special relativity effects such as doppler effect.

8 Disclaimer

The entire code was written in C++ on the MAC OSX 10.8.2 operating system with 2GHz processor Intel Core i7. There is a Makefile included in the submission that can be used to compile the code in the folder.

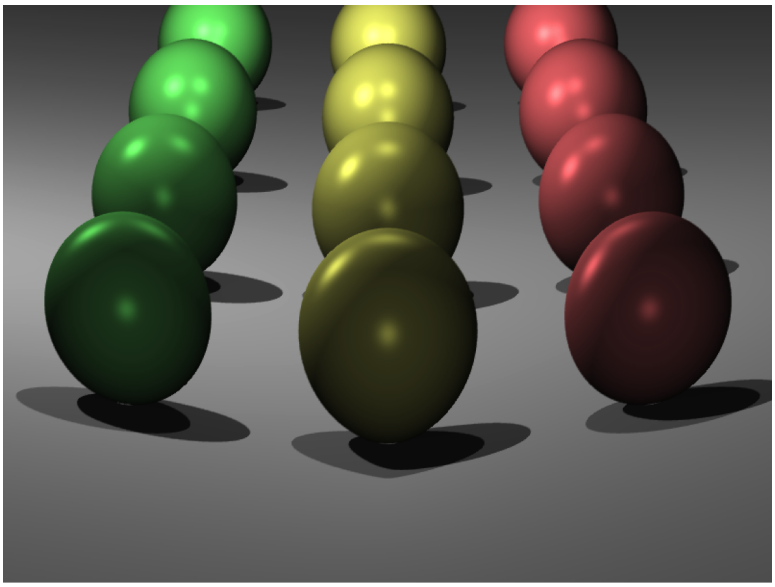

Figure 9: $\beta = -0.66$.

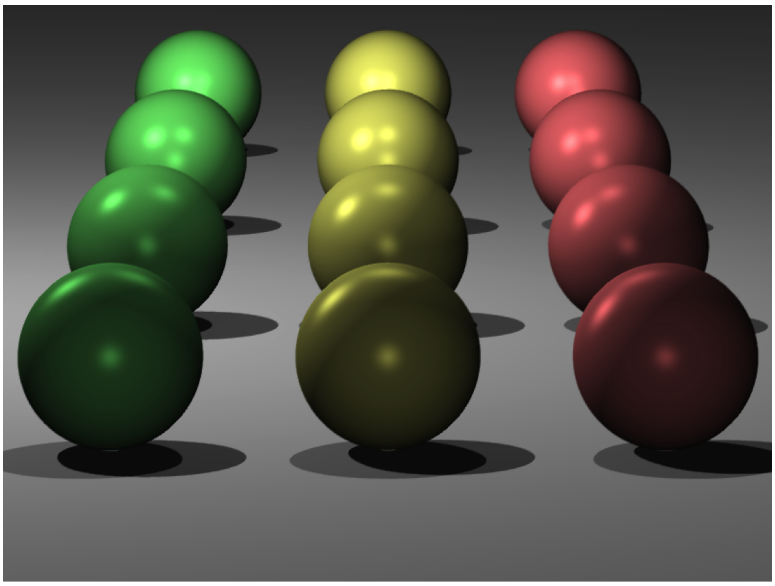

Figure 10: $\beta = 0$.

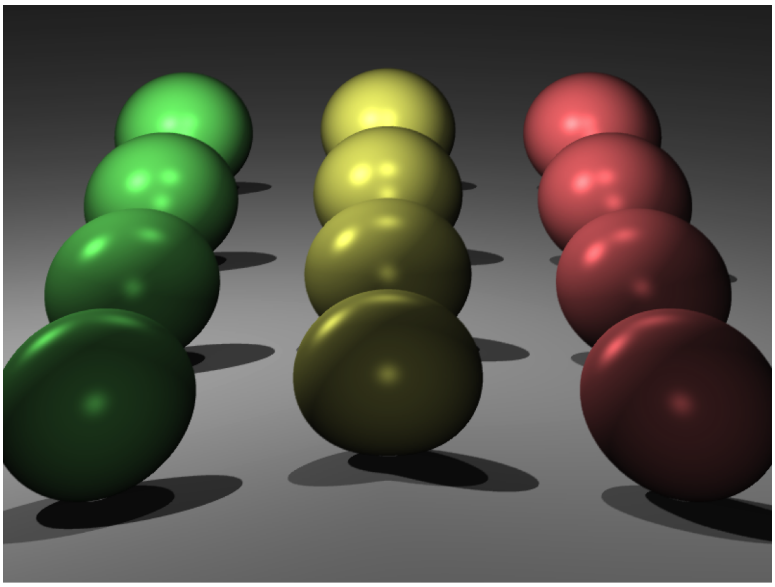

Figure 11: $\beta = 0.66$.

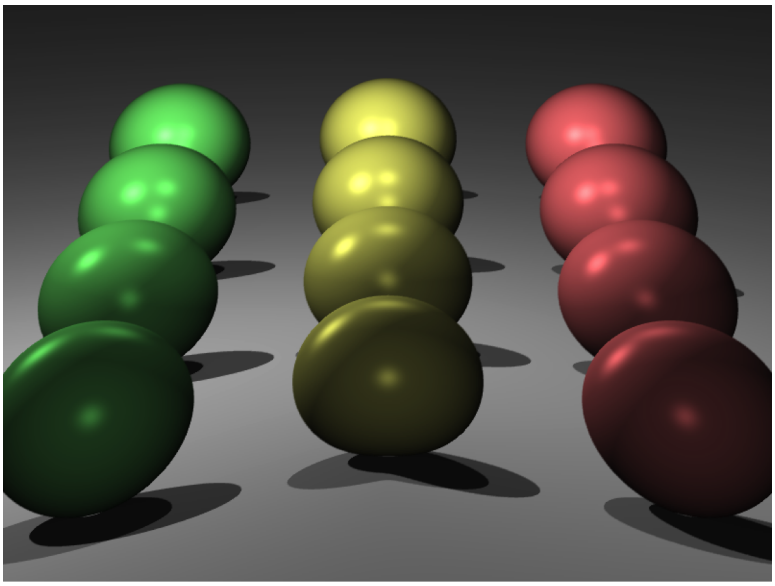

Figure 12: $\beta = 0.75$.

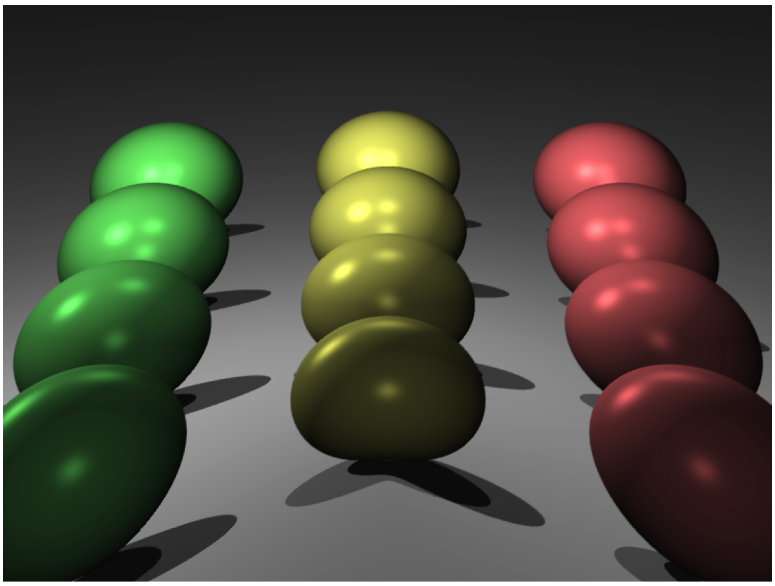

Figure 13: $\beta = 0.90$.