LuoLei “Larry” Zhao (netID: lzg431)

Tumblin

EECS 395

Project B: Raytracing System

User Guide:

1. Opening the program

To open the program simply **click** on the HTML file labeled “ZhaoLuoLeiprojB.html”. You will see a two side-by-side screens. The left one has a Perspective viewport already loaded, and the right one will contain your ray traced image.

2. Moving around

All movement functionalities are made with the keyboard. To move around, simply use the **W,A,S** and **D** keys. The **W** and **S** keys will make you walk forwards and backwards, while the S and D keys will make you strafe left and right. To survey your surroundings by looking around. You can use the **U,H,J,K** keys, which function similarly to the WASD keys. All instructions are written underneath the Canvas screen.

3. Move and adjust the light

There are two lights in the Scene. The first light is a “headlight” which follows your character. The second is positioned in the world. You can move around the light using the **O,K,L and ;** keys, which are used in a similar manner to the W,A,S, and D keys. By moving the center of the light, you can view how shadows change as the lights interact and overlap.

4. Changing the Scene/Turning Lights on and Off

To change between the two scenes, you can simply scroll down and find a set of buttons that allows you to switch between the two available scenes. Additionally, clicking the “Toggle Headlight”, and “Toggle Other Light”, will switch the two lights on and off alternatively.

5. Adjusting Aliasing/Number of Reflections

In addition, you can change the number of reflections by simply clicking on the button. The webpage will then cycle through all the reflection options, with the current number of reflections being displayed. You can similarly cycle through the options for Anti-Aliasing dimensions and jittering using the provided buttons

Code Guide:

Basic Ray Trace Structure:

1. General Side by Side Structure

Our code is organized into two separate files. The file traceLighting.js holds all code regarding the Ray Tracer, as well as a few bits of general management code, such as the HTML button and keyboard callbacks. In contrast, the file VBOBuffer.js holds almost all information regarding the webGL display, with a little bit of extra code that sets up the buffer for the Ray tracer texture. Although many variables are actually shared between both displays, the two file format allowed for easy organization of code.

2. Camera manipulation and movement

Both the Web-GL display and the ray tracer displays share identical variables for all position and camera information to ensure that the images match. When adjusting the orientation of the Ray-Tracer camera, I ended up creating my own setLookAt function, which mirrored the one provided in the quon-matrix library very closely. This function modified the u, n and v vectors of the camera so that when an image was created, the rays will be cast towards the right general direction before pixel offsets are applied.

3. Anti-Aliasing

Even without aliasing, our ray caster iterates through all of the pixels of our display and casts a ray for every pixel. Increasing the aliasing parameter causes the ray caster to perform multiple casts per pixel. The multiple casts have their own small offsets from the original ray, based on the current set dimensions of the Anti-Aliasing. Turning on jitter will also some randomization to the offsets, which can help create better samples. After all samples are made, the program averages their results as the color displayed.

4. Shapes

In addition to the extra shape described in the section below, our raycaster allowed for the rendering of Spheres, Cylinders, and a ground plane. The ground plane, being a single, static object that exists at a consistent location, had its position calculated by directly evaluating whether the ray cast vector was pointing downwards, and if so, setting the intersection point as the hit position. All other shapes first transformed the ray vector with the inverse of the current transformation matrix for that shape. After that, all calculations were made on a unit model using the transformed ray. For the sphere, the 2 chord method was used to calculate the hitpoints, which allowed for saving a lot of time. In addition, the normal was calculated as the ray going form the sphere's center directly to the hitpoint. The cylinder was calculated using an implicit equation for the sides of the cylinder. After that, the top and bottom of the cylinder were calculated by finding whether the ray crosses a "ground plane" at the top and bottom, and if so, whether it hits within the radius of the top/bottom of the cylinder.
4. Switching Scenes.

5. Lights and Shading

All lights positions were saved into the same values for both the Ray tracer and the Web-GL tracer. After the rays were traced, a hit list was created. The member inside the hit list with the shortest distance from the origin is sent to the findShade function, and the cHit object is passed. This function then iterates through all lights to calculate the light vectors that may strike the object. If an object intercepted the light ray, that particular light ray would be aborted for this hitpoint, resulting in a shadow. If light does manage the strike the object, the light vector would be used in a standard phong lighting calculation, utilizing information such as the normal and hitpoint from the chit object.

6. Reflections

In addition to the phong calculations made by the light ray, if the number of reflections was set to a number greater than 0, a second ray would be cast from the hit point in the findShade function, with its direction being the calculated reflect ray based on the origin eye ray and the normal. All additional ray traces will be passed a "numReflect" parameter that decreases with each recursion. After the findShade function is finished, it will return the reflection color. All reflection calculations are then combined with the original color with proportions depending on the object's reflection constant value.

EXTRAS

Extra Shape

4. Cube

In addition to the 3 shapes mentioned in the section above. I also created a simple cube shape. As with all shapes, I first transformed the ray with the inverse model matrix, before running the implicit equation. While more efficient methods likely exist, I simply tested 6 individual equations for the six planes of the cube, and kept only rays whose hitpoints hit within the bounds of the cube. Then I took the hitpoint with the lowest t0 value among the six planes.

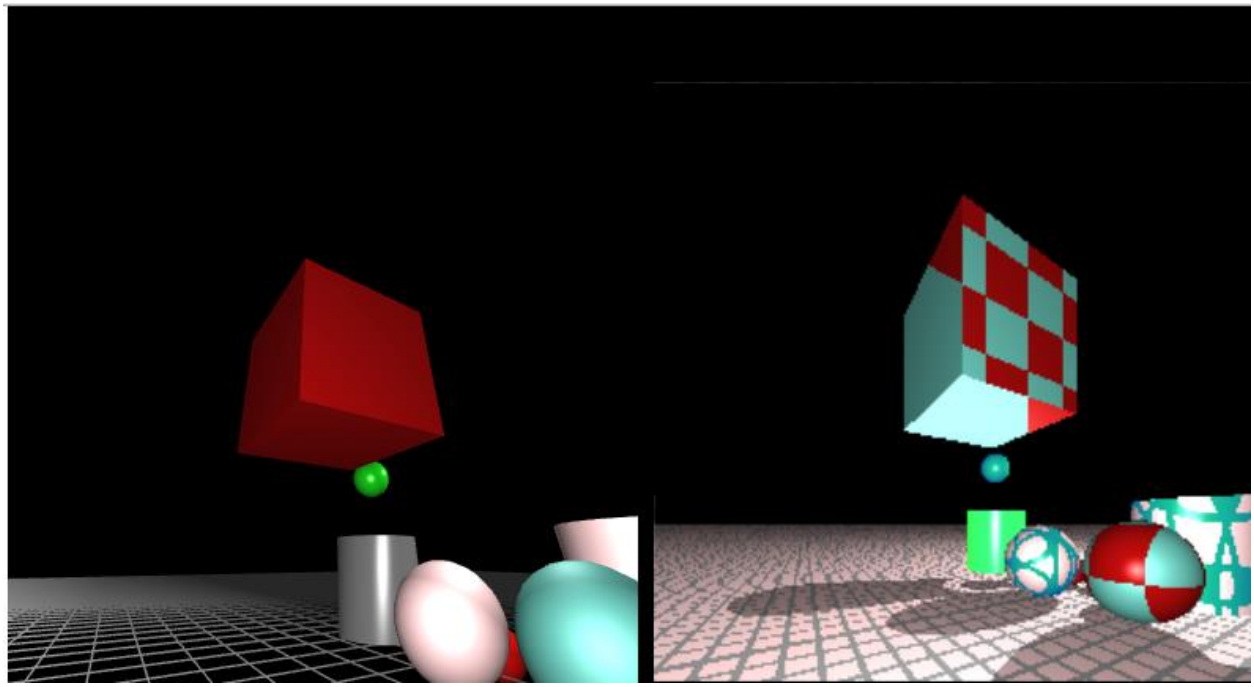

Figure 1: Raytraced Cube with procedural texture.

Refractions:

In addition to reflections, I also created a very simple refraction framework based off of Snell's law. Certain objects were assigned the property "transparent". If true, in addition to the reflection rays, an additional refraction ray was cast. The ray's angle was modified based off of the normal at the hitpoint. And the refracted ray's color was combined with the object's phong lighting color, as well as the reflection color.

Procedural Materials:

Certain objects in the scene had materials whose textures were procedurally generated based off of various functions passed to the object. All CGeom objects had a special variable "matlfunc" which could be set to a procedural material. Upon a ray hitting the object, the CGeom object will check if a procedural function exists, run the function, and return the result as the material.

1. Checker Board

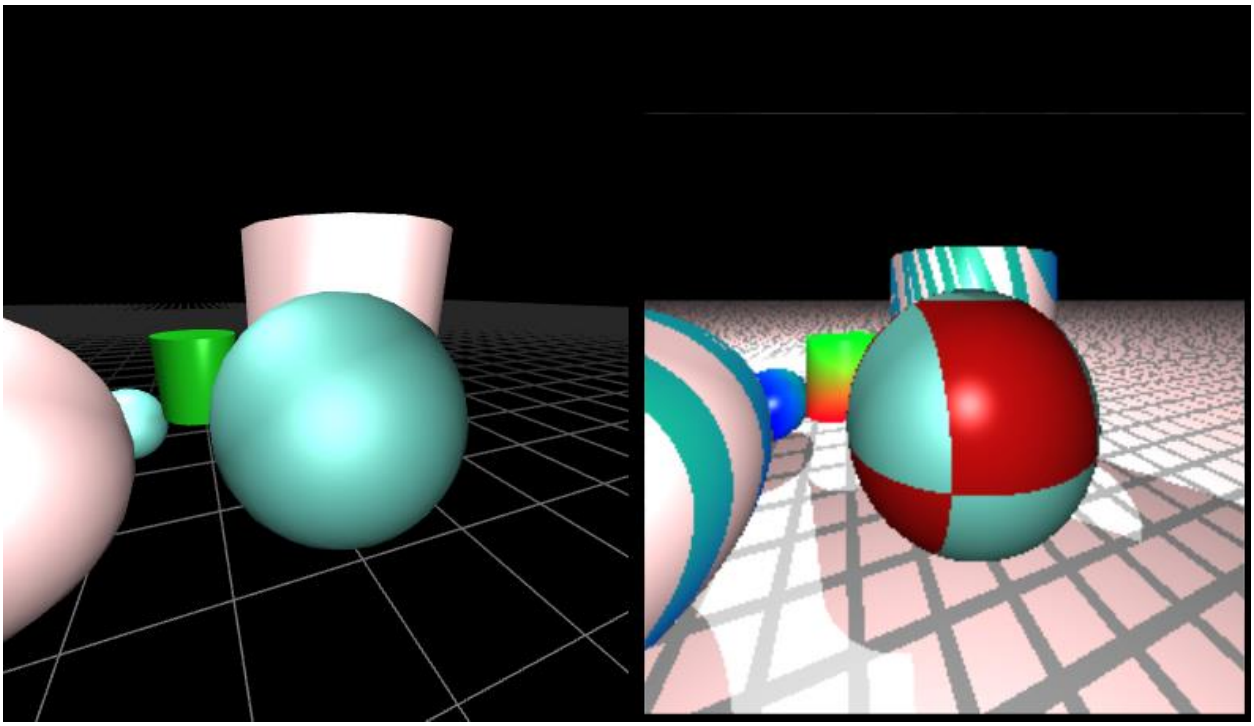

Figure 2: Checkerboard sphere

The first procedural object was a simple checkerboard design. This material took a hitpoint parameter and took the floor of the x,y, and z coordinates added together. It then floored all values 2 and returned one material if the result was 0 and another one if otherwise.

2. "Banded"

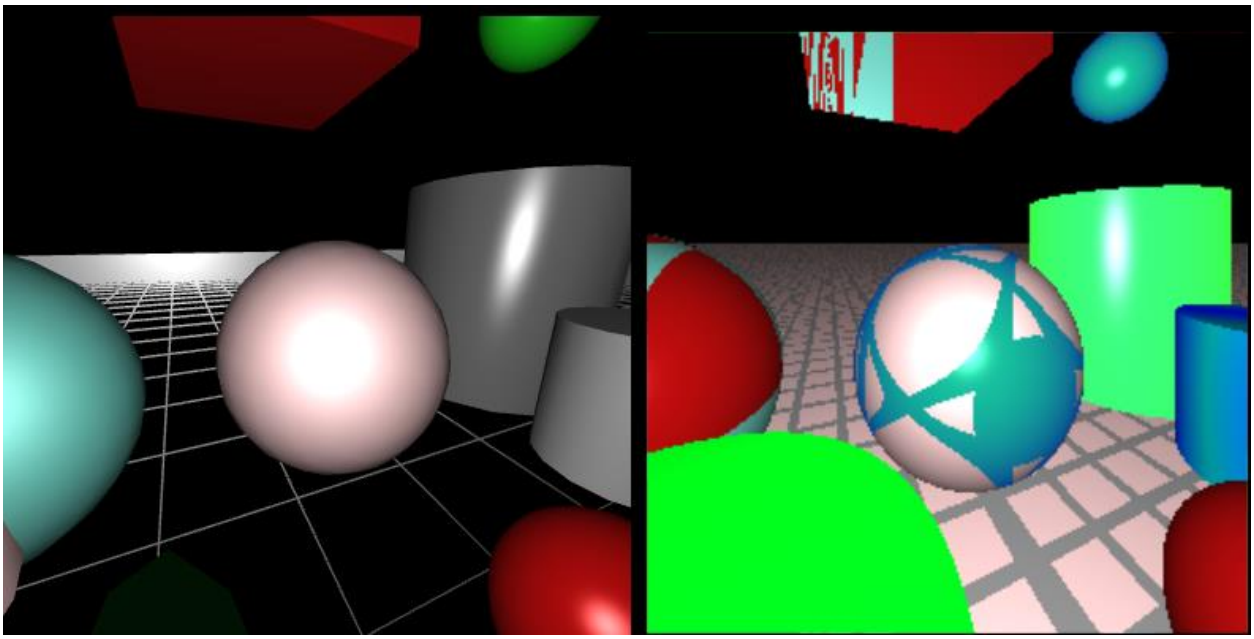

Figure 3: Banded Sphere Procedural texture

Taking a similar idea from the checkerboard design. This procedure took a pre-arranged list of "spots" as well as the hitpoint of the ray. The "spot" positions were first transformed by the object's transformation matrix to make a similar texture appear regardless of the object's location. The procedure sums the distance of the hitpoint from all of the spots, and returns one material if the sum of the distances exceeds a certain threashold, and a separate material if it does not.

3. Gradient

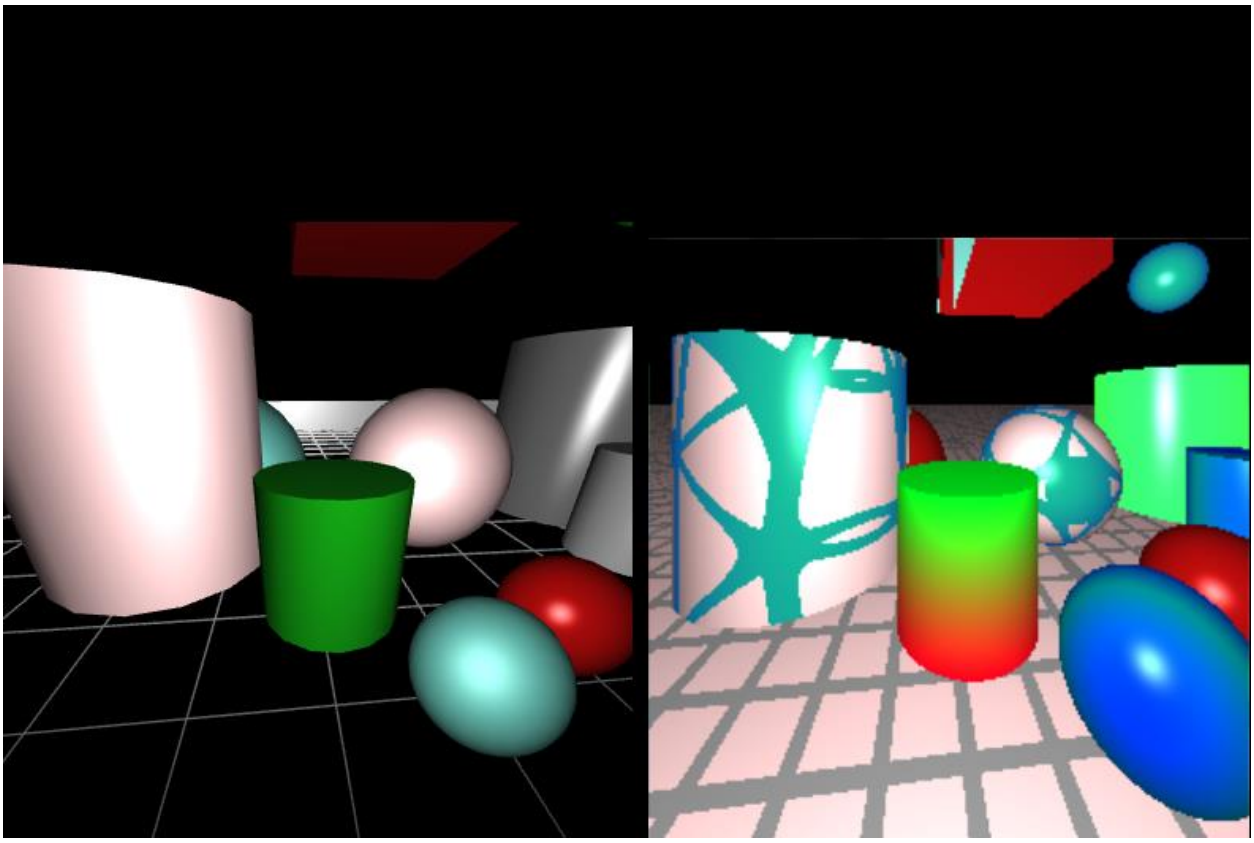

Figure 4: Gradient Procedural texture on cylinder between two banded texture objects

In contrast to the discrete materials of the previous textures, I wanted to create something smoother. I created a texture that steadily increased the green diffuse and ambient values of the material as the z axis of the hitpoint increased, and increased the red value as the z value decreased. This created an interesting watermelon like gradient.

4. "Marble"

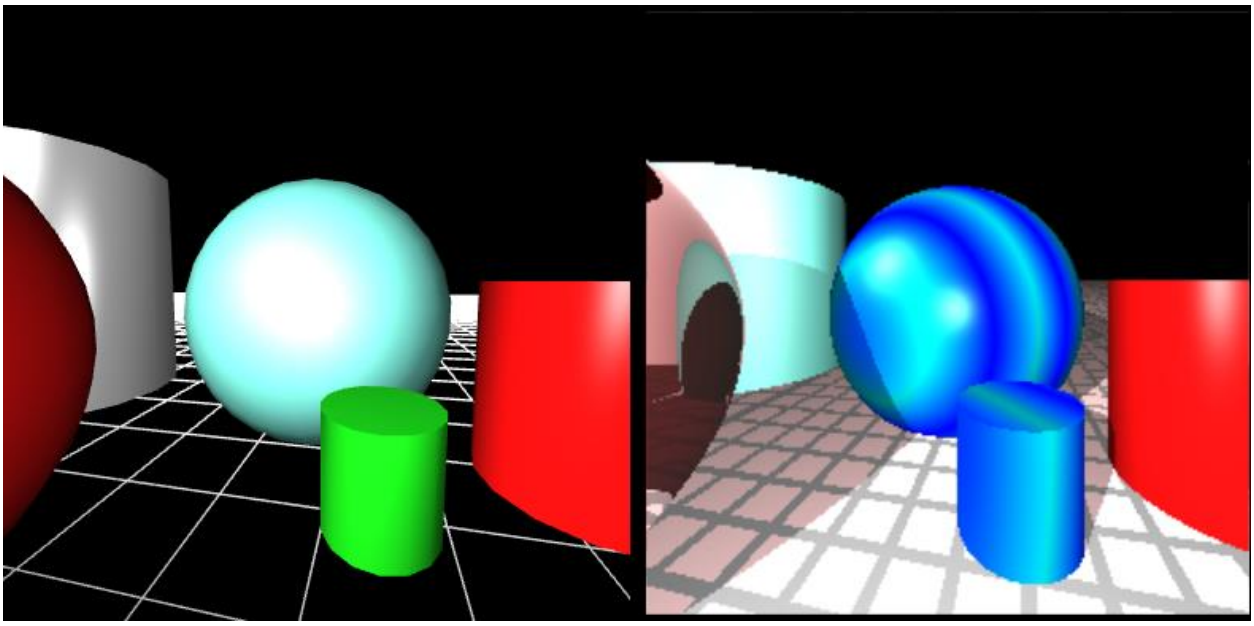

Figure 4: Marble Sphere and cylinder

Combining all techniques from above, I decided to create a design that based itself off of a set list of positions like in the "banded" texture but still be smooth like the gradient. The result was a beautiful glass marble texture. I was able to achieve the smooth gradients by first modding the distances by 1, and then setting the result as the Math.abs(dist – 0.5), allow the texture to smoothly transition from one color to the next.

Additional Figures:

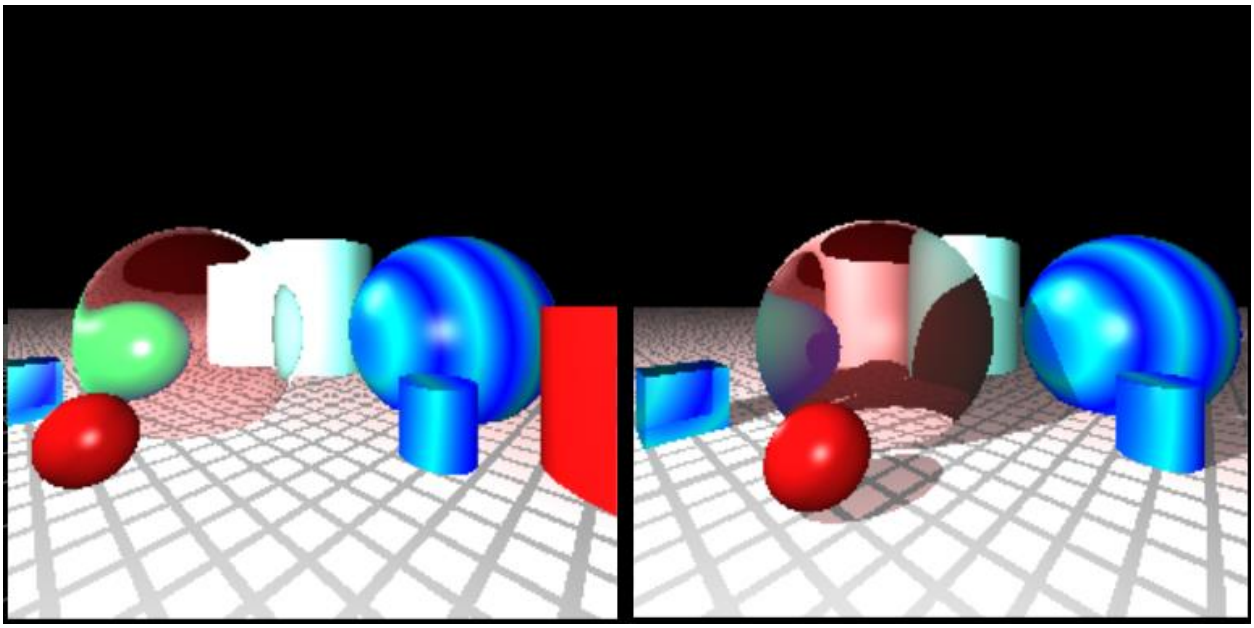

Figure 5: Before and after view of a similar scene with and without shadows

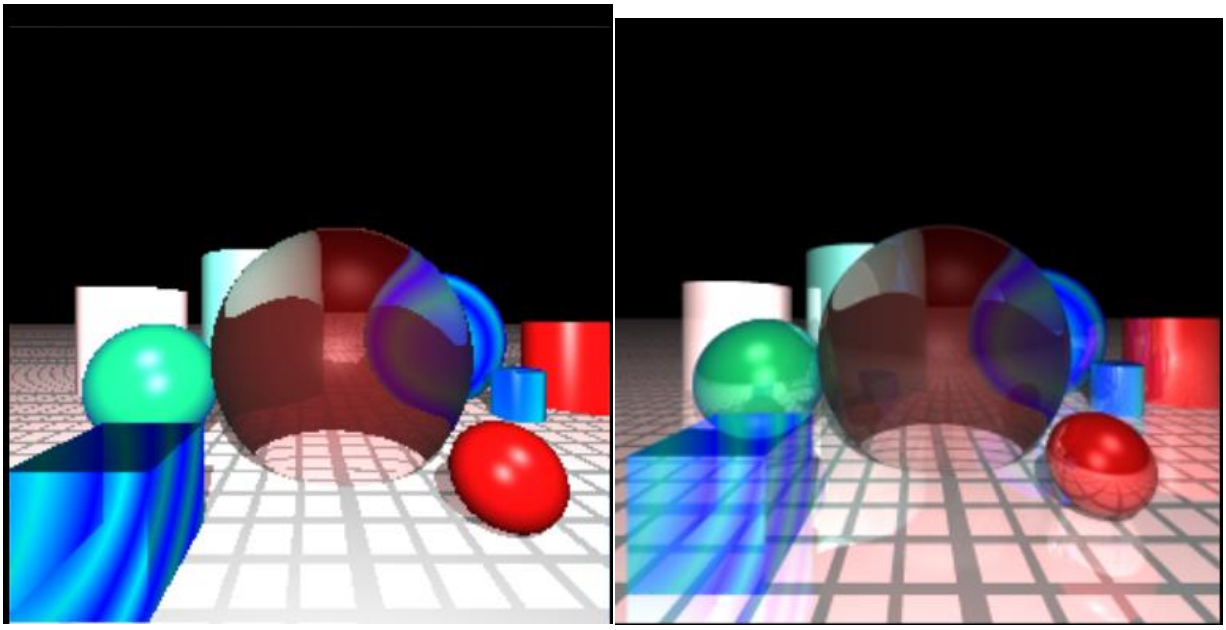

Figure 6: Before and after view of image with 4x4 anti-aliasing. 3 reflection depth and jittering applied.

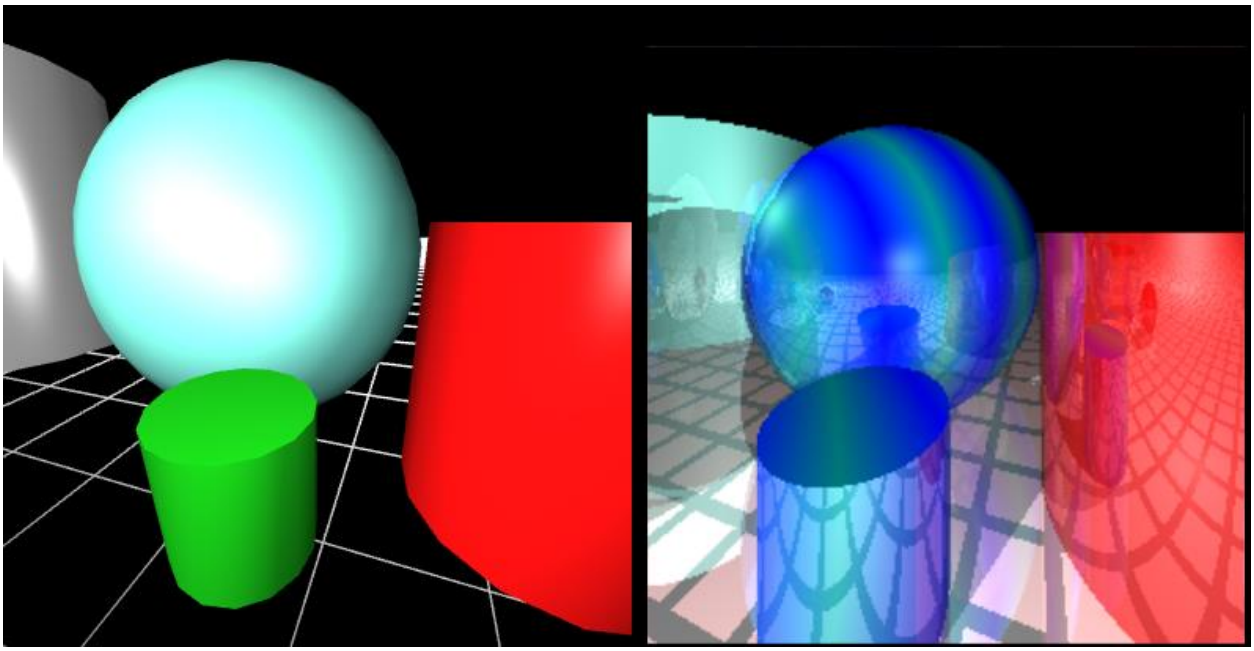

Figure 6: A clear view of intereflections between objects. Here you can see the red cylinder and blue cylinder reflecting off of each other.