COSC363 Assignment 2

Duncan Tasker: 46250511

June 2024

1 Scene Description

The average time to render the frame with anti-aliasing enabled was around 165ms, or reduced 60ms without anti-aliasing at a resolution of 800×800 rays. This is possible due to spreading the primary rays across 25 threads. Overall the quality of the image is quite good with the only major artifact appearing when there are multiple objects of differing transparencies between a point and the light source, as the ray will assign the shadow color according to the first object it intersects rather than continuing to trace the ray towards the light source to see if an object will completely occlude the ray. This manifests as bright spots in shadows depending on the placement of objects. This can be avoided with careful scene composition. The scene can be seen in figure 1.

2 Extra Features

2.1 Cone and Cylinder

Using the cone-ray intersection formula and cylinder-intersection formula I constructed the basic form of the objects. Combining this with the normal methods allowed both objects to have the same properties as both sphere and planes (excluding texture support). To generate the caps on these shapes it was a simple matter of additionally testing an intersection with a plane located at the bottom of the shape (or both top and bottom) bounded by the radius of the cone or cylinder. It should be noted that the cone is casting a shadow correctly, it is just hidden behind the geometry of the cone and as such is not visible. It can be partially seen in the reflection on the blue sphere behind it.

2.2 Refraction

In order to implement refraction in a manner that was compatible with the recursive tracing implementation as well as remaining fully thread safe, I adjusted the function signature for the `trace()` to pass not only a ray object, and the current step count, but also the refractive index of whatever substance the current ray is in. By default this value is 1 as the ray initially passes through air,

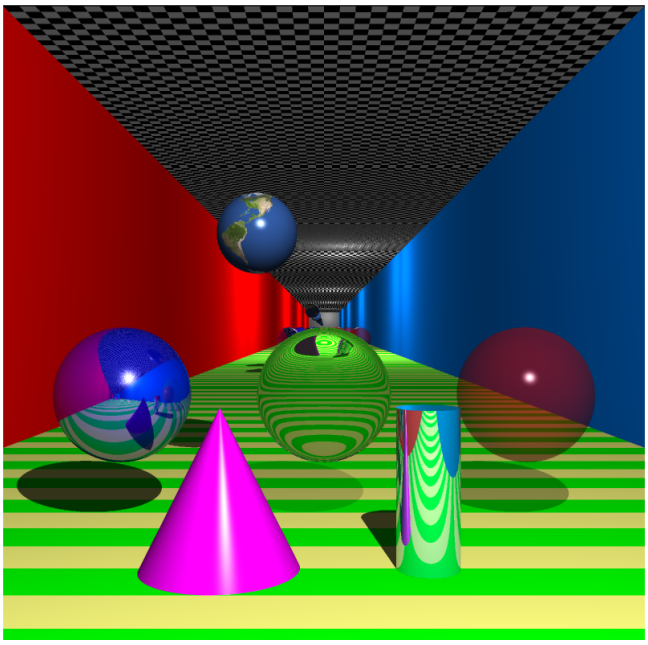


Figure 1: Shapes showcased inside 6 plane room

however this is updated to the refractive index of the object the ray intersects with upon collision. This method allows the function to remain recursive as it can always calculate the refractive ratio according to equation 1 since both refractive indices are known.

$$n_{\text{relative}} = \frac{n_{\text{current}}}{n_{\text{object}}} \tag{1}$$

2.3 Multiple reflection

In the scene both the front plane behind the camera and the back plane behind all the objects are perfect mirrors with a reflective index of 1.0. This creates the "infinite corridor" effect with a maximum recursion depth of 10 steps. All optical effects like reflection, refraction, and transparency were all implemented recursively so in the final image the "depth" of the corridor would be affected by the objects which it is interacting with along the way.

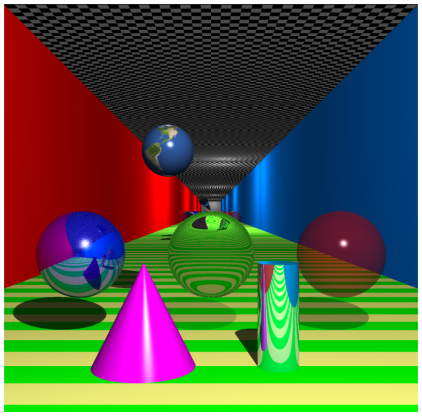
2.4 Basic Anti-aliasing

To achieve this effect for each primary ray is perturbed by a small amount into each of the four quadrants and traced normally. The amount each ray is perturbed by is defined by equation 2 which is then added to the primary ray

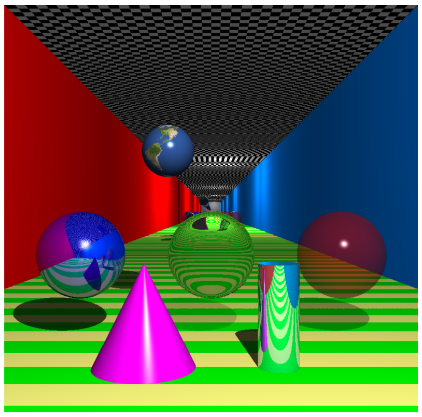
to determine the final direction of the anti-aliasing rays.

$$\text{perturbed ray} = \begin{bmatrix} dx * \left(\frac{(x_{max} - x_{min})}{\text{NUMDIV}}\right) * \text{offset} \\ dy * \left(\frac{(x_{max} - x_{min})}{\text{NUMDIV}}\right) * \text{offset} \\ 0.0 \end{bmatrix} \tag{2}$$

Where dx and dy are variables iterated in nested for loops in the range $[-0.5, 0.5]$ with a step size of 1 moving the direction of the ray into each quadrant. NUMDIV represents the number of rays along the X and Y axis of the final image, and offset a constant set at 0.025 which was used to adjust the strength of the anti-aliasing effect. The resultant colors of the four cells are then averaged to smooth the image and reduce the amount of jaggedness on edges and artifacts on refractive surfaces. The results with anti-aliasing can be seen in figure 2a, compared to without in figure. 2b.



(a) Scene with anti-aliasing



(b) Scene without anti-aliasing

Figure 2: Comparison of Anti-aliasing

2.5 Textured sphere

To add textures to a sphere I dynamically assign `UV` coordinates to the sphere allowing the texture to be projected properly. To achieve this, we pass the current hit coordinates to the `getColor()` method of the sphere object, and compute the relative position on the sphere and perform some trigonometric functions to compute the actual texture coordinates and return that color.

3 Extra Extra Features

3.1 Boundary Volume Hierarchy

In order to reduce the number of intersection tests in the scene I constructed a bounding volume hierarchy tree to hold each of the scene objects which could be traversed to find ray intersections, resulting in an asymptotic complexity for each ray reducing from $O(n)$ to $O(logn)$. This is accomplished by storing increasingly small axis-aligned bounding boxes (AABB) containing the scene objects in a binary tree.

To implement this required the addition of an AABB parameter to the `SceneObject` class. This required each shape to have it's own instance of calculating a bounding box which must be called during the object's constructor but allowed for much simpler construction of the BVH tree. To build this tree, the `BVH` class is passed a list of all the scene objects. Then algorithm 1 is used to construct the tree.

Algorithm 1 Creating the BVH Tree

function BVH CONSTRUCTION(objects)
    Set node's AABB as the union of all objects contained in the node
    if Number of objects is less than leaf threshold (2) **then**
        Set node as leaf
        return
    end if
    Find largest axis of the AABB's bounding box
    Sort all contained objects according to their position on this axis
    Find the object closest to the midpoint of this axis
    if This middle object is the first or last object in the list **then**
        choose the middle indexed object instead to avoid empty nodes
    end if
    Call function recursively on the left and right children, passing the left child all object from the start of the list to up to the midpoint object, and passing from the midpoint object to the end of the list to the right child
    return
end function

To test intersections with the BVH tree, a first in, first out stack is used following algorithm 2. This ensures that once a point of intersection is found, nothing farther than that point could be tested, resulting intersection tests on average per ray in complex scenes.

A notable downside of this method is that in this implementation, where scene objects are simple shapes that have their own defined intersection formulae, the cost of intersecting an AABB is equal to the cost of 6 separate ray-plane intersections, so until the scene contained a significant number of objects – by my testing around 150 to 200 spheres – the cost of traversing this BVH tree was greater than simply intersecting every object in the scene. This would not

Algorithm 2 Intersection Testing using the BVH Tree

```
1: function BVH Intersection(p0, dir)
2:     Add root to stack
3:     while stack is not empty do
4:         pop item off stack
5:         Test intersection between ray and this node's AABB
6:         if There is no intersection point(s) found then
7:             Perform no further processing and continue to next iteration
8:         end if
9:         if The current closest intersection point is closer than the point between the ray and the current AABB then
10:            Perform no further processing and continue to next iteration
11:        end if
12:        if The current node is a leaf node then
13:            for Each object contained by the node do
14:                Test intersection between ray and object
15:                Store current hit distance and object index
16:            end for
17:        else
18:            Add the current node's left and right children to the stack
19:        end if
20:    end while
21:    return The closest hit coordinates, distance, and object index
22: end function
```

matter in real world implementations where models are constructed of hundreds of thousands of triangles, however for this simple program it proved detrimental to performance. In the scene show in in figure 1, activating BVH intersection testing increased frame times from 165ms to 830ms. I left using BVH intersections as an optional toggle-able parameter by pressing 'b' on the keyboard while the program is running.

3.2 Multi-threading

Implementing this was crucial reducing the draw times of the final image. There were several changes that needed to be made to the original code to allow me to leverage multiple threads in drawing the scene. Firstly, all the state for colors such as on the planes had to be encapsulated into the shape objects themselves, and not checked for in the trace method as this could cause race conditions and flickering colors on objects in the scene. This was relatively simple to implement as it only required expanding the `object.getColor()` to accept the hit coordinates as a parameter which could be used to calculate the final color of the object at that location.

In addition it required a global data structure which could hold the ray objects such that the threads could return values that could be used when drawing the scene. This was accomplished through the `RayBatchFactory.h` and `RayBatchFactory.cpp` files which handle the allocation and freeing of all the primary rays used in the scene.

To split the rays between threads, each thread is passed a ray wrapper struct which contains an array of pointers to the ray objects stored in the global data structure as well as the number of rays the thread is responsible for. The thread then iterates through the array, tracing each one before returning. When rendering the scene at 800×800 rays, with 25 threads operating concurrently, each thread is responsible for $25,600$ rays resulting in an approximately 7 to 8 times reduction in render times from approximately 1230ms per frame operating in single threaded mode, compared to 165ms per frame.

4 Build Commands

To build this project run the make.sh script with the clean argument like so from the base directory:

```
./make.sh --release --clean
```

Which will clean any previous CMake caches and build from a clean directory. Be sure to use the release flag, as this will apply compiler optimizations and significantly reduce frame draw times. After this has finished, the executable can be found in:

```
./bin/RayTracer
```

5 Student Declaration

I declare that this assignment submission represents my own work (except for allowed material provided in the course), and that ideas or extracts from other sources are properly acknowledged in the report. I have not allowed anyone to copy my work with the intention of passing it off as their own work.
Name: Duncan Tasker
Student ID: 46250511
Date: 31/5/24

6 References

The main resource used for implementing the BVH structure was this 2012 presentation by Magnus Andersson which can be found here All other textures, and resources were taken from class lab and lectures.