REAL-TIME NON-EUCLIDEAN RAY TRACER ILLUSIONS

Michael Johnson (mikejohn), Lingshu Tang (lingshu)

Stanford University
CS148 Introduction to Computer Graphics and Imaging
Instructor Zahid Hossain

Goal: *Our proposal was to develop a real-time ray tracer to maneuver through a non-Euclidean space scene and use illusions to display the unusual behaviors of this environment. The first technically challenging aspect was to achieve real-time ray tracing at a reasonable frame rate. The second challenging problem was to construct our non-Euclidean space in a mathematical manner and to figure out how objects and distance are affected by the non-Euclidean space.*

1. Introduction

Unlike the traditional three-dimensional Cartesian space, a non-Euclidean space does not follow all five of Euclid's postulates. The two most studied non-Euclidean spaces are hyperbolic and elliptical space (see Fig. 1). Ordinary objects are distorted due to distance variation characterized by the non-Euclidean space. For instance, a sphere will be stretched in the z-axis while being compressed in the x-y axis in the center of a hyperboloid space. Our goal was to utilize a real time ray tracer to view the misshapen objects and stretch or compress distance defined by our non-Euclidean space.

2. Approach

Inspired by Varun Ramesh's project Dygra [1], we wanted to create a space where locations and distances defy our traditional understanding. Initially we thought adding portals to our regular Euclidean space is enough to define our non-Euclidean space, similar to the game Portal. Although implementing portals into our world would make our space non-Euclidean, we believe it wasn't technically difficult or interesting. We can simply change the view of our camera based on the camera's coordinates as it moves through the scene. Thus, our interest shifted to creating a mathematical model of a hyperboloid space where the outline of spheres or cubes will be defined by the space. With that in mind, Mike began by implementing the framework to conduct real time ray tracing while Ling worked on defining the hyperboloid space.

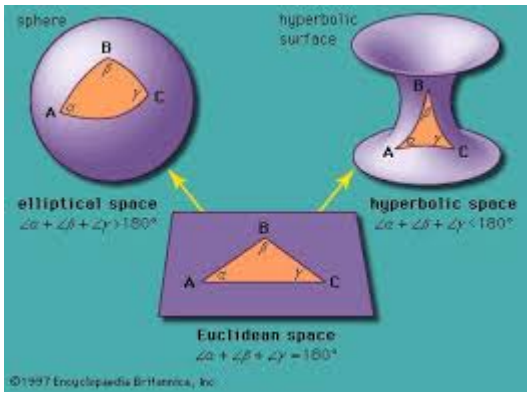

Figure 1. Example of Euclidean and Non-Euclidean Space

3. Ray Tracing Strategy and Challenges

To get started, Mike's initial strategy was to build a real time ray tracer on top of CS148's HW4. Mike used the library Simple DirectMedia Layer version 2.0.4 (SDL2) to handle keyboard / mouse inputs and render the images as high as 60 frames per second [7]. This game loop allowed us to move through the scene relatively fluidly and view the scene's shapes by changing the camera's position, heading, and pitch. Our first scene consisted only of a simple sphere.

Mike was able to implement a real time ray tracer on top of HW4 – with one small caveat. The objects within the scene were in homogenous coordinates, and was not able to transform the view back to world view due to difficulty uncovering the underlying assumptions made by the CS148 staff for HW4. Thus our objects are stretched in the z-direction [2]. Since the transformation matrices were not easily accessible and because our machine's local CPU was not be able to handle the loads involved with real-time ray tracing anyway, Mike went on to develop a custom ray tracer in OpenCL (C language).

4. Non-Euclidean Space Implementation and Challenges

Ling's goal was to understand the mathematics behind hyperboloid one sheet space and how distances are varied within the space; based on the curvature of the hyperboloid. Also, he wanted to convert Euclidean distance into hyperbolic distance for the scene. Unfortunately, after reading several articles regarding the math behind hyperbolic space [3], we quickly realized that implementing the space mathematically in computer graphics isn't something achievable within weeks. Ling went on to try the Euclidean distance to hyperbolic distance but he had little success (see Fig. 2). The team then decided to pursue a different approach. Our new approach is to generate a hyperboloid with HW4's code with a new intersection function (see Fig. 3). After that, Ling added a sphere into the hyperboloid where it is distorted due to the confinement of the hyperboloid space [2].

not
not

$$\sinh d_h = \frac{\sqrt{|D_{pq}^2 - A_{pq}^2|}}{\sqrt{1 - D_{pq}^2}\sqrt{1 - D_p^2}}$$

where $D_{pq} = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$ is the Euclidean distance between the points,

$D_p = \sqrt{x_p^2 + y_p^2}$ and $D_q = \sqrt{x_q^2 + y_q^2}$ are the Euclidean distances from the origin and,

$$A = x_p y_q - x_q y_p$$

Figure 2: Algorithm for Euclidean distance hyperbolic distance

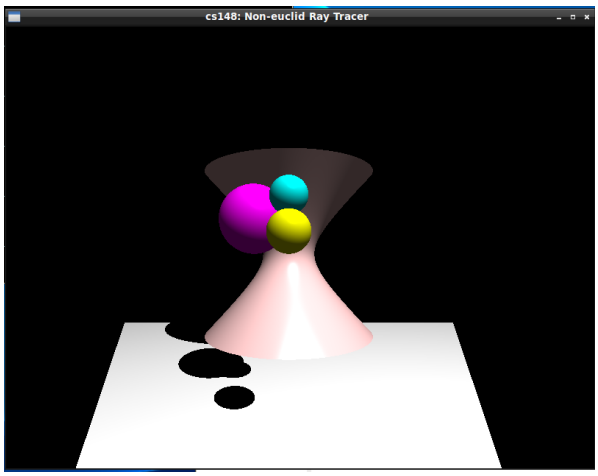

Figure 3: non-Euclidean Ray Tracer (First Attempt)

5. Ray Tracer in OpenCL

Our first attempt to develop a real-time ray tracer relied entirely on the CPU for computations and rendering. This significantly constrained the program's ability to compute the path of rays beyond a simple a scene as shown in Figure 3. Additionally, our ray tracer was far from real time, rendering below 2 frames per second. Thus, we determined that utilizing the machine's GPUs for computations would drastically improve the frame rate of our ray tracer. We researched potential solutions for GPU use including CUDA, OpenGL compute shaders, and OpenCL, ultimately deciding to use OpenCL due to Mike's proficiency writing in C.

Unsatisfied with our first attempt, Mike implemented the real-time ray tracer using OpenCL and a C++ program with SDL2 and OpenGL. As before, Mike used SDL2 to set up the main game loop at a maximum of 60 frames per second, handle all keyboard / mouse events, open a window, and render the scene. Mike then used OpenGL to maintain and update matrices associated with applying user inputs to the position, heading, and pitch of the camera, or the position of the objects in the scene. We wanted the objects to be defined in the world coordinate system, so we passed the view matrix to the kernel to put the camera in the world coordinate system before the ray tracer began computations.

The most challenging aspect of this project came in writing the custom ray tracer in kernel code (C), which does allow for recursive functions, printing to the screen, or other basic libraries, and then implementing the communication between the GPU and the CPU using OpenCL's C++ Bindings. We drastically underestimated the amount of time it would require to complete this task as debugging kernel code is extremely difficult without the ability to easily print to screen.

Mike built the ray tracer to account for reflections, shadows (also handling shadow acne), anti-aliasing, Phong shading, and rendered a few basic shapes including spheres and planes [4]. Mike implemented the first scene consisting of four checkered patterned planes for a ground plane, ceiling, and two tilted walls, with four spheres, two of which are moving back and forth along the y axis. Fig. 4 shows a snap shot of this scene, and a demo has been uploaded to YouTube [5]. Even with this simple scene, we only achieved a maximum of about 10 frames per second, and turned off anti-aliasing and specular lighting to do so, highlight the computationally complexity required for real-time ray tracing.

6. Non-Euclidean and illusions

After this first scene was created, we went on to attempt to implement an illusion using non-Euclidean geometry. This illusion would be a tunnel which from the outside looked of normal length, but once the user was in the tunnel, seemed longer than it appeared. This effect would be achieved by utilizing aspects of non-Euclidean geometry and expanding space within the tunnel.

Mike and Ling both implemented the rendering of cuboids (rectangular prisms), using those as building blocks

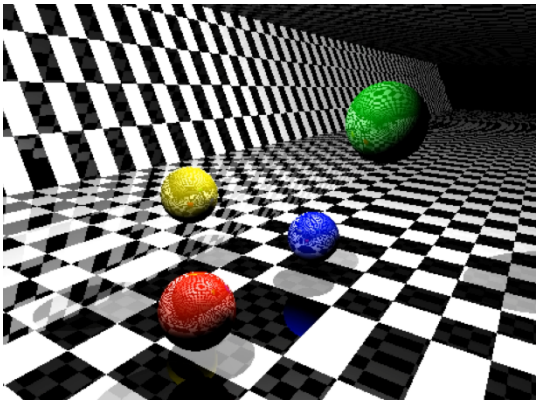

Figure 4: Scene one of GPU driven ray tracer

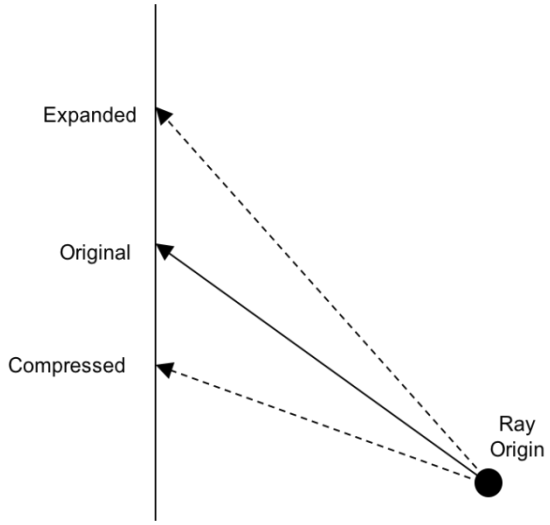

Figure 5: Depiction of ray direction scaling. To create non-Euclidean effect, must multiply desired direction of ray by scale.

to create the tunnel. Mike then implemented the tunnel expansion by scaling the camera ray direction in the +z direction, creating the illusion of an expanded tunnel for the non-Euclidean space effect. Fig. 5 displays how a ray could be scaled to create this effect.

Finally, fig. 6 and 7 show two separate snap shots of the tunnel both outside and then inside of the tunnel, and we uploaded a demo to YouTube [6]. In the demo, we see the user enter the non-Euclidean space where the tunnel seems to extend and the time to travel through the tunnel is much longer than the time to travel around the tunnel. The abrupt change from Euclidean to non-Euclidean space is a result of some challenges discussed in the next section.

7. Challenges and Future Work

We had several challenges along the way to come this far. First, we believe this project was quite large in scope attempting to implement a real-time ray tracer using GPUs while doing so in a non-Euclidean environment. Each of these pieces took a significant amount of time to understand and then implement to the point where we ran out of time to fully explore a non-Euclidean space with other aberrations and portals.

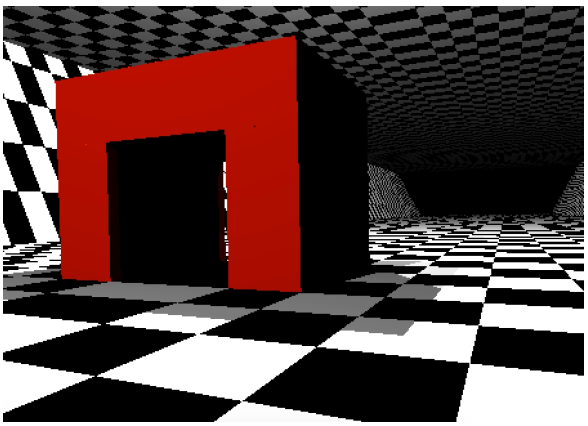

Figure 6: Tunnel illusion from the side.

Figure 7: Tunnel illusion from within the tunnel. In this example, the camera ray directions are scaled so the tunnel seems longer than it appears from the outside.

Second, the implementation of the ray tracer from scratch in kernel code was more difficult than anticipated. Code executed on the kernel does not allow for recursions, does not easily enable printing to the screen which made debugging difficult, and disallowed the use of libraries to, for example, multiply matrices by a vector. Additionally, misunderstood at the time, as the kernel code grew in size (e.g. adding more code for the tunnels), the frame rate of our ray tracer slowed down significantly. One deficiency in our code's structure was generating our scene from the kernel code itself rather than generating the scene in the C++ program (like in HW4), and passing the details of the scene to the kernel. The ray tracer would have had a much improved frame rate if done so.

The final challenge was the execution of the tunnel illusion. Although, the process to expand the space within the tunnel was not as difficult to implement, rendering the tunnel such that when the user views into the tunnel from the outside, the inside of the tunnel already looks expanded. This is the reason for the abrupt change from Euclidean to non-Euclidean space as the user enters the tunnel, the latter which appears to extend the length of the tunnel. Since we needed to keep track of when a ray enters the tunnel and exits the tunnel, we had to include other shapes in the scene to mark these points. This required writing a second intersect function for a shape in which instead of determining the point where the ray intersected with the outside of the cuboid for example, the ray intersected with the inside of the cuboid. Furthermore, we had some trouble making the boundary shapes translucent with our current ray tracer which blocked the entrance to the tunnel.

In the future to improve the project, we could speed up the ray tracer by leaving only essential functions in the kernel code and implementing an algorithm such as bounded kd-trees to improve the processing of reflections. Furthermore, if time permitted, we would have liked to fix the few issues with the current tunnel illusions as well as create portals or other aberrations, for example, anything with the region of a shape will have the laws of non-Euclidean geometry applied to the demonstration. Also, using a computer with a faster graphics card would have helped as well.

8. Conclusions

To implement a real-time ray tracer is not easy. To implement a real-time ray tracer that can trace non-Euclidean space is an even harder challenge. Despite the fact that we were unable to quite achieve our original proposal, we learned various aspects of computer graphics and still accomplished a lot with our project. From how to use OpenCL to understanding the math behind the non-Euclidean space in graphics, we overcame many obstacles. Furthermore, we challenged ourselves in terms of quality and spent a substantial amount of time learning our respective areas.

Acknowledgements

All of the CS148 staffs who were kind to help us throughout the project, and all the YouTube tutorials on OpenCL.

References

[1] Ramesh, Varun
http://www.varunramesh.net/projects/raytracer

[2] Project first attempt: https://youtu.be/maCNCl4Y-a8

[3] Caroline Series, *Hyperbolic Geometry*, http://homepages.warwick.ac.uk/~masbb/Papers/MA448.pdf Refer to introduction page vi for hyperboloid model equation and Chapter 1.1.1 Cross-ratio and transitivity of Aut(C) for Euclidean to hyperbolic distance transformation on Page 4-8

[4] http://www.scratchpixel.com

[5] Demo of first GPU scene:
https://www.youtube.com/watch?v=LgSueFRbaCM

[6] Demo of tunnel illusion:
https://www.youtube.com/watch?v=heJZ3g1F94k

[7] SDL (v2) Docs: https://wiki.libsdl.org/FrontPage

Other Tutorials and Documents

[8] OpenCL Docs: https://kosobucki.pl/cl_doc/index.html

[9] SDL and Ray Tracing:
https://www.youtube.com/watch?v=W2QrXv2yZhE

[10] Object intersection equations:
http://www.realtimerendering.com/intersections.html

[11] World, View, and Projection Transformations:
http://www.codinglabs.net/article_world_view_projection_matrix.aspx

[12] OpenCL tutorial:
http://simpleopencl.blogspot.com/2013/06/tutorial-simple-start-with-opencl-and-c.html