Ray-Tracing as a Gameplay Element in Interactive Media

I: Abstract:

Ray tracing is a technique that simulates the behavior of light to render highly realistic images. With the advent of real-time ray tracing hardware, this computationally expensive procedure is no longer limited to pre-rendered contexts such as film and animation. The video game industry has adopted ray tracing in its pursuit of increasingly realistic graphics. However, little, if any, attention has been paid to how realistic light simulation might affect gameplay in addition to increasing visual fidelity. In this project I explore this concept by creating a demo for a simple ray traced game in which the player must find passwords in order to escape rooms. Passwords are hidden and obscured with optical tricks, like reflection and refraction, that players must see through to complete the levels. While this demo does not use real-time ray tracing, the concepts on display are easily extensible to a real-time context. Even with non-real-time rendering, significant effort was spent implementing acceleration techniques and engineering an interactive framerate. This project was largely successful, as 4 distinct levels were created, each showcasing a different approach to utilizing ray tracing as a gameplay element. Given this demo's small scope and limited access to specialized hardware, the ability to create this unique experience is encouraging. Further investigations should continue to prioritize and improve rendering speed while increasing visual fidelity. Going forward, there is still ample room to explore this concept further and introduce this novel gameplay idea into a real-time context.
II. Introduction:

Ray tracing is often considered the holy grail of graphics [1]. By simulating light rays as they travel through space, scenes can be rendered with extreme fidelity, reproducing the myriad effects and interactions that work together to achieve photorealism in a rendered image. Ray tracing, a time-consuming and often brute force method, dominates in settings where ample time and computing power can be given to producing realistic effects, like rendering an animated film. In real-time settings such as video games and simulators, such an intensive process has long been impractical, if not impossible, to implement while maintaining an interactive framerate. This is no longer the case. Within the past year major graphics card companies have announced and released GPUs with dedicated hardware for real-time ray tracing. These cards allow games to utilize ray tracing while maintaining standard frame rates of 30, or even 60, frames per second. However, much of the focus on this technology has been on its use for improving visual quality in games. Light from explosions can be reflected off of cars, torches can provide soft and dynamic shadows, and water can realistically mirror the game world above it, to name a few examples. Little attention has been paid to using ray traced graphics to create unique gameplay experiences. The conversation about ray tracing has completely centered cosmetic improvement, with no discussion of how this technology can be used to influence players' goals and how to achieve them. This project is a proof-of-concept demo for a game where players use ray-traced graphics to help solve a short series of simple puzzles.

The stated purpose of this project was to create a game with ray-traced graphics to demonstrate how realistic lighting effects and optical tricks could be incorporated into a game as gameplay elements. While this project would not utilize real-time raytracing, it would seek to demonstrate the fitness of ray tracing as a gameplay element, in real-time contexts or otherwise. The reason for using a non-real-time setting was due to lack of familiarity and, more importantly, access to new and expensive
hardware. Thus, the game would comprise of a player controlling a camera in a scene, providing still renderings each time the camera moved. It was proposed that players would control this camera via either buttons on the screen or the command line. The game itself was to be comprised of a series of carnival-funhouse levels with elements including mirrors and refractive lenses, with the player operating a robot that would navigate each room. Finally, it was discussed that each playthrough might produce a report to keep track of player movement, and analyze how players move through the levels and interact.

III. Process:

In order to create this demo, I began by creating a simple raytracer, capable of rendering spheres and triangles. This formed the foundation for the entire project. In order to maximize speed the project is written in C++, and relies on two libraries. Simple Directmedia Layer (SDL2) was used to create and manage the game window, as well as handling input events for player interaction. OpenGL Mathematics (GLM) streamlines the various mathematical operations required for ray tracing, and is particularly useful for its vector class. In order to facilitate level creation, scenes are read from .json files using the RapidJSON library. With a functional ray tracer that could read geometry, lights and camera position from a file, I added the ability to move and pivot the camera during runtime. At this point it became necessary to work on accelerating the rendering process. With this in mind, I implemented a uniform grid. With this acceleration structure, rays do not check for collisions against each piece of geometry in the scene. Instead, the scene is divided into a three-dimensional grid, and the ray only checks for collisions in cells it crosses. To provide an even more dramatic speedup, the main render loop was parallelized using OpenMP. At this point a simple scene (6 walls, several spheres, no 3D models) with no anti-aliasing took 1-2 seconds to render at a resolution of 640x480. In order to increase the speed with which a player can interact with the game, I lowered the resolution to 160x120, decreasing the render
time dramatically to between 0.08-0.20 seconds with a simple scene. The player can still view render the view at original resolution with a single input to get a detailed look at the scene. In order to create more interesting scenes, I added the ability to load 3D models from an .obj file. I also added textured rectangles, using the CImg library to read .bmp files. Finally, in order to give the player a way to beat the levels, I added the ability to enter a password. If the player finds and enters the correct password, the level ends. During the entirety of this process I consistently met with Professor Rushmeier to ensure that the project remained on schedule and on goal.

Throughout the course of development I encountered many roadblocks. Many, of course, were confusing but simple, requiring a one-time fix. For example, the game window would enter a "Not Responding" state after several seconds of rendering. Even when the render was finished and control returned to the window, SDL2 failed to render the pixel buffer to the screen. After probing the SDL forums, I discovered that the library wants to constantly check for events from the user. Failing to do so will result in a "Not Responding" state, which eventually leads to the undefined behavior of the window refusing to update. In order to fix this, I added a timer into the render loop and check for events every second. Tonemapping was a more complicated issue that required more experimentation to solve. There was an early issue when I absentmindedly capped a pixel’s RGB values at 1.0 within the render loop, flattening the highlights of the rendered image and impeding any effort to tonemap in a post-processing step. I fixed this problem by allowing a pixel’s color values to exceed one, then fitting them into the range [0, 1] after the render loop. This produced its own issues, however, specifically that the presence of a bright object in the scene would drastically darken the surrounding geometry. Once that object left the frame the rest of the scene would become dramatically brighter. In order to fix this, I included two tone mapping methods into the post-processing step. The first was a relative approach, a filmic method by John Hable [2] that produces a much more even toned, if slightly dark, image. Once this was done I used a global approach of pinning the midpoint of pixel intensity to an RGB in the
middle range of $[0, 1]$. This produces a much clearer image with minimal variance in brightness from frame to frame.

By far the largest roadblock during this project, and the focus of much of the development time, was the speed of rendering the scene. Without a sufficiently fast render, all sense of interactivity in the game is lost. Before any optimization a simple room of 6 walls and 4 spheres could take 5-8 seconds to render. This time would of course scale with more objects. With anti-aliasing (an option that I implemented but chose not to use for speed-up reasons), time increases exponentially. Several measures were taken to accelerate rendering. First, as mentioned above, a uniform grid was implemented. This acceleration structure is useful, but difficult to tune. If the grid resolution isn't right, the cost of navigating the grid can outweigh the benefit of fewer collisions. I tried to use an algorithm to analytically determine the grid resolution based on the distribution of objects in the room, but this proved unwieldy and unhelpful. In the end, I hand-picked the uniform grid resolution for each level. Furthermore, without enough geometry in the scene the grid serves little purpose. For example, in a simple room with only 6 walls, each of which spanning the length of an entire side of the grid, the grid offers little to no advantage. In fact, I found that using a grid to check for light visibility was too costly with only a small number of lights, and opted for the brute force method of checking every collided object against every light. Finally, any object that moves between frames either needs to be placed again in the grid, or placed in all cells so that it can always be collided with. In practice, however, only one object has this property (the camera sprite), so this had minimal impact on performance.

I also used multithreading with OpenMP to reduce render time. This created a drastic speedup (the machine used to develop this project could handle 4 threads). Parallelism, however, rarely just works without issue -- several objects presented data hazard in a parallel environment. When a ray collides with a textured rectangle, for example, the rectangle stores the UV coordinates of the point of the collision to use later
when determining pixel color. Having every thread try to access this value presents an obvious hazard. One solution was enclose the entire block from where the collision occurs to where the surface color is determined in a critical section, ensuring only one thread can access it at a time. For such a significant portion of the rendering cycle, however, this would eliminate much of the benefit of parallelism. A better solution was to store 4 copies of the UV variable so that the threads do not have to compete for access.

IV. Results:

The result of this project is an interactive demo, featuring 4 levels, that demonstrate how ray tracing can be used to help solve puzzles in a game. The final game implements a ray tracer that can render triangles and spheres, simulate reflection and refraction, and can display textures, all specified from a .json file. Players can move freely around the world and angle their camera to look around. The world is rendered at a very low resolution by default, but players can produce a detailed render on command. In each level players will look for a password which, upon successful entry, will end the level. The players controls are as follows:

<table>
<thead>
<tr>
<th>WASD</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrow Keys</td>
<td>Look Around</td>
</tr>
<tr>
<td>Space</td>
<td>Hi-Res Render</td>
</tr>
<tr>
<td>Enter</td>
<td>Enter Password</td>
</tr>
</tbody>
</table>

Levels are each contained in a separate folder. To play a level, run `./game` with the appropriate folder name as an argument. For example, `./game 2` will play the level in
folder 2. Levels are found in a `scene.json` file contained in each folder. The program itself operates in three main steps plus cleanup:

- **Setup:** SDL2 creates a window for pixels to be rendered to. `scene.json` is loaded. Triangles, spheres, textures and the camera are all created and stored within a scene object. A uniform grid is then created and all objects are placed into it. A buffer of unsigned 32-bit integers is created to be filled with pixel values in an ARGB format.

- **Render:** A main render loop operates on each pixel, casting a ray through each pixel. Each ray traverses the uniform grid, checking for a collision against every object in the cell the ray is currently intersecting. Geometry is implemented as a variety of classes derived from a `Shape` parent class. Each class defined an `intersect` method for determining ray collisions, a `normal` method for finding surface normal, and a `surface` method for determining pixel color upon collision. After the main render loop is completed the pixel buffer undergoes post-processing for tonemapping. The buffer is then rendered to the screen.

- **Input:** A check is executed for any keyboard events that may have occurred during or since rendering. Any key press that would change the camera view or resolution triggers a re-render of the scene from the new position or resolution (going back to the render stage). If enter is pressed, the program will accept text input as a password attempt.

- **Cleanup:** The program releases all allocated resources and closes the game window upon exit.

A glass sphere at two different resolutions. **Left** 160x120. **Right** 640x480.
The final result differs from the original proposal in several ways. Currently no method for reporting or analyzing playthroughs exists in the code. Adding this would be simple, as the code base is organized in such a way that player position is easily available at each step, and could be reported with the inclusion of several additional lines of code. That being said, it was apparent during development that such a system would be of minimal utility to designing levels. Levels were largely designed around a single concept, so players had little trouble figuring out where to navigate. Furthermore, because players do not interact with levels, but rather observe and analyze them, it was more useful to watch players play levels in person while they share their thoughts aloud. This is obviously an unscalable approach to testing, but for such a small demo it was sufficient. This was far from the only realization made during development. Indeed, this project became less of an in-depth exploration of ray tracing, as was originally proposed, but more of an exploration of how ray-tracing and interactivity could integrate. For example, the concept of rendering implicit surfaces as fun-house mirrors was discarded in favor of producing models of these surfaces using triangles, which would serve the

One sphere reflected with two mirrors
same function and allow for easier design iteration. Furthermore, when the player is given control of the camera, certain considerations must be made that may not be necessary when the designer has full control. Triangles, for example, should be treated as double-sided geometry so that players do not walk behind an object only to find it has disappeared. Most differences from the original proposal resulted from streamlining the project due to time restraints. Bump mapping, for example, was originally planned, but was cut because it largely adds aesthetic value to the ray tracer without adding gameplay functionality.

V. Conclusion:

This project was largely successful in accomplishing its original goal -- to produce an interactive ray traced demo that showcases how realistic light effects might be used for gameplay purposes. In any areas where the original vision was compromised, such sacrifices were made to narrow this focus on gameplay, while shedding extraneous or cosmetic features. In the future, this project could expand to include such cosmetic features. Bump-mapping would add a level of visual flair to scenes. If further acceleration were pursued path-tracing could become a viable option to produce far more realistic scenes. Adding more features of a standard game engine, however basic, would also expand the possibilities of mechanics that could be introduced. Simple collision detection or a system of levers and switches could go a long way in immersing a player within the game world. Finally, the most important next step would be the extension of this concept into the realm of real-time ray tracing on dedicated hardware. This would allow a complete investigation into capacity of ray tracing to serve as a gameplay element in modern video games.
VI. Sources:
