1 Purpose

The goal of this project is to render a nice scene from a puzzle game I completed recently called Monument Valley. The art style of the game is very simple so I will add textures, lighting, and better shadows to the scene, while still having something that looks like a level from the game. Doing so will allow me to better understand and demonstrate some of the concepts I've learned in the course.

2 Statement

Monument Valley is a puzzle game where the player needs to help the main character, Ida, through a series of levels comprised of geometric landscapes, often with optical illusions and seemingly impossible geometry. The levels are architecturally complex with hidden paths and rotating platforms that the player must manipulate to help Ida reach the end of the level.

I have attempted to model and render a scene that has a similar geometric and blocky style that would belong in the levels of the game, with mostly rectangular objects and buildings. To incorporate the concepts learned in the course, I added textures, bump mapping, better lighting, reflective and refractive objects, and an interesting depth of field effect. I will use constructive solid geometry to create interesting shapes that I can place in the level (in the game Ida places shapes such as Cubes with their centers hollowed out in a certain way to end the level, so I can include shapes like those). I have also added a Mesh object in the scene to demonstrate Phong shading.

3 Manual

The basics of running the ray tracer with existing .lua files can be found in the readme, as well as decriptions of the new lua commands added to access to creates scenes with the objectives I've implemented.

4 Implementation

4.1 Lua Extensions

4.1.1 gr.render_advanced

Usage:

$$gr.render\_advanced(scene, filename, width, height, eye, view, up, fovy,$$
$$ambient, lights, focalLength, apertureSize, superSample, useParallelRendering)$$

Identical to the provided gr.render but with additional parameters controlling focal length and aperture size for depth of field, and booleans to control super sampling and parallel rendering.

4.1.2 gr.nh_rectangle

Usage: $gr.nh\_rectangle(name, pos, size_x, size_y, size_z)$ Similar to gr.nh_box, but the three dimension can be specified separately.

4.1.3 gr.area_light

Usage: $gr.light(x, y, z, r, g, b, c0, c1, c2, radius)$. The first three parameters are identical to the default gr.light, the radius is used to control the area of the light for soft shadows.

4.1.4 gr.advanced_material

Usage: $gr.advanced\_material(kd, ks, shininess, reflectivity, transparency, indexOfRefraction)$. Identical to the provided gr.material, but with additional parameters for controlling reflectivity and transparency and index of refraction.

4.1.5 gr.textured_material

Usage:

$$gr.textured\_material(texture\_fname, ks, shininess, reflectivity, transparency,$$
$$indexOfRefraction)$$

. Similar to gr.advanced_material, but the texture is loaded from a file and used in place of the kd parameter.

4.1.6 gr.bump_textured_material

Usage:

$$gr.bump\_textured_material(texture\_fname, normalmap\_fname, ks, shininess, reflectivity, transparency,$$
$$indexOfRefraction)$$

. Similar to gr.textured_material, but contains a normal map as well.

4.1.7 gr.csg

Usage: $gr.csg(name, operation)$. Creates a CSG node with the given name and operation. Operation can be "union", "intersection", or "difference". It expects two children to be added to the node, which can either be primitives or other CSG nodes.

4.1.8 Global Constants

SKYCOLOUR
VOIDCOLOUR
SHADOW_PETURBATION (For shadow calculation and recursive rays)
MAX_DEPTH (maximum number of recursive rays)
DEPTH_OF_FIELD (always set to true, turn on and off by setting aperture size to 0)
DOF_RAY (number of rays to shoot from each aperture)
SOFT_SHADOWS (always set to true, turn on and off having area lights)
SHADOW_RAYS (number of rays to shoot for each area light)

4.2 Mirror Reflections and Refractions

In my objective list I have these as two separate objectives, but their implementation ended up being related so I will discuss them together. Mirror reflections are perfomed on materials with non-zero reflectivity. The direction of the new ray is calculated as by the CalculateReflection function in Raytracer.cpp. The formula is simply:

$$I * 2 * n * dot(I, n)$$

Where I, n are the incident and normal vectors.

Refraction is performed by calculating the new direction of the ray by Snell's law. I used the following formulas derived by Benedict De Greve:

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

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

Both of the above are done in Raytracer.cpp in ComputeRefraction and ComputeReflection functions.

One thing I didn't think I would need but ended up being useful was Fresnel. It ended up greatly improving my refraction. For refractive objects the Fresnel coefficient is used to calculate the amount of refraction and reflection based on the angle the ray hits the surface. The Fresnel coefficient is calculated by the following formula:

$$F_\parallel = \left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}\right)^2$$

$$F_\perp = \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2$$

$$F_R = \frac{1}{2}\left(F_\parallel + F_\perp\right)$$

All these values were already calculated in CalculateRefraction so it was simple to create a new FresnelCoefficient function in the Raytracer.cpp file.

4.3 Texture Mapping and Normal Mapping

Texture and Normal mapping involved creating the TextureMap.hpp/cpp and NormalMap.hpp/cpp classes. TextureMap is a general class and I've implemented ImageTextureMap as a class for image textures. Both the Texture and Normal map classes have a GetKd/GetNormal method which returns a colour/normal based on u, v coordinates. These classes are added as a point in the PhongMaterial class. When performing a lighting calculation in the PhongMaterial.cpp class, it checks if a texture or normal map exists before applying it.

One problem I encountered with normal mapping was that the normals defined in the normal map all face in the positive Z direction, but my surface could be facing in any direction. Thus I also needed to modify my Intersections.hpp/cpp class to contain a tangent, and I modified my Cube and Cylinder primitives to calculate a tangent as well. Then I used the Normal and Tangent to calculate a Bitangent (cross product), and used these three vectors to create a matrix that transform the mapped Normal to the correct space. This was done in the PhongMaterial.cpp class.

u, v coordinates for rectangualr prisms were easy as I simply took the coordinates on each face and applied a modulus of 1. For cylinder the vertical direction up and down the size face was done in a similar way for the v coordinate, but the distance along the circumference was used for the u, and calculated using circular coordinates. Simlar u, v on a sphere was calculated using the azimutha and polar angles.

4.4 Anti-Aliasing

Simple Anti-aliasing was implemented by shooting rays in the four quadrants of each pixel. This is implemented in the SuperSample function in Raytracer.cpp. I also implemented a simple form of adaptive anti-aliasing, where the colour in each quadrant is compared to the centra colour. If the difference is large, I shoot four more rays in that quadrant. The difference between colours is calcuated using the "redmean" which was defined in an article on CompuPhase.

$$\bar{r} = \frac{C_{1,R} + C_{2,R}}{2}$$

$$\Delta R = C_{1,R} - C_{2,R}$$

$$\Delta G = C_{1,G} - C_{2,G}$$

$$\Delta B = C_{1,B} - C_{2,B}$$

$$\Delta C = \sqrt{\left(2 + \frac{\bar{r}}{256}\right) \cdot \Delta R^2 + 4 \cdot \Delta G^2 + \left(2 + \frac{255 - \bar{r}}{256}\right) \cdot \Delta B^2}$$

4.5 Phong Shading (Normal Interpolation)

I got Barycentric Coordinates for free inside of Mesh.cpp when I implemented the Möller-Trumbore algorithm as part of A4. Using the u, v coordinates, I simply return a weighted sum of the normals of the vertices of the triangle.

$$intersection.normal = glm :: normalize((1 - u - v) * m\_normals[tri.n1]$$
$$+u * m\_normals[tri.n2] + v * m\_normals[tri.n3])$$

4.6 Constructive Solid Geometry - Simple and Complex

I was able to successfully implement complex CSG so I will skip the discussion on simple CSG. As described in the lecture notes, and the Piazza post by Dr. Mann, complex CSG is implemented by performing operations on sets of lines. I've added a new method to the SceneNode.hpp class called CSGIntersect. In the base class SceneNode.cpp implementation this method simply groups the result of its children and returns one big vector of line. In the Geometry.hpp/cpp class, it also calls CSGIntersect on its primitive, before calling it on its children. In the Rectangular Prism, Sphere, and Cylinder primitives a pair representing the line segment of itersection is returned for the parent GeometryNode to collect. This was simple to implement as the shapes are convex and calculating the regular intersections already implements the needed code for finding the line segment intersections. The case of exactly 1 intersection is ignored since I don't really see why it would be very important to be accurate around corners and edges when performing boolean operations where there is likely lots of overlap.

When a ray is initially traced, all intersections are regular intersects until intersect is called on a CSGNode node. The CSGNode call CSGIntersect on its two children and recursively gets two sets of lines. CSGNodes can be the root of a tree of any size but in my implementation it must be a binary tree.

The operations can be performed as follows (implemented in the CSGNode.cpp file):

4.6.1 Union

The union operation is the simplest, my code appends the second set of lines to the first.

4.6.2 Intersection

The intersection operation simple loops over all pairs of lines from the first paired with the second set. Each time it performs an intersection between two individual lines. This involves taking the max starting point and min ending point of the two lines. If the max starting point is greater than the min ending point, then the lines do not intersect. If they do intersect, the intersection is added to the result set.

4.6.3 Difference

The difference operation is the same as intersection except it subtracts the second set of lines from the first. Again it looks at two lines one at a time. The algorithm for performing the difference between two lines simply looks at the different cases for where each line starts and ends. The only thing to be careful of was to flip the normal in certain cases.

4.6.4 Merging overlapping lines

In the Union operation it may be obvious that this will create unecessary overlapping lines. For this reason in the SceneNode.hpp/cpp class I've added the SortAndMergeSegments method. The algorithm comes from a LeetCode question I solved before https://leetcode.com/problems/merge-intervals/description/. It requires sorting the lines by starting point, which then makes it easy to merge overlapping lines from start to end.

4.7 Soft Shadows

Soft shadows are implemented first by modifying the light class (Light.hpp/cpp) to contain a radius field, and have an instance of a nh sphere as a private member field. The rest of the code is mainly in the Raytracer.cpp file. A shadow calculation is made by calling CalculateShadow. If the current light is a point light (radius 0), then the shadow will be calculated by simply casting 1 ray directly towards the light. Otherwise a fixed number of rays defined by SHADOW_RAYS is cast towards the light, and the number which hit is used to integrate the area of the light visiable to the surface.

To generate the random rays, I first create two vectors that define a plane perpendicular to the light direction. I do this by taking the cross product of the light direction with the unit vector (0,1,0) to get 1 vector, and taking the cross of this new vector with the light direction to get the other vector. Then I calculate a random radius (bound by the real radius of the light), and a random angle. The random angle is used to convert from circular to cartesian coordinates, which then define a perturbation of the two vectors generating above, which now defines a random point on the area light. Although I'm using a sphere to determine if these new rays hit the light, the light being emitted is simply treating the light as a circle perpendicar to the point on the light when viewed from any angle, and light is being emitted equally from all points.

4.8 Depth of Field

Depth of field is implemented by simulation an aperture, shooting multiple rays from random points in the apertures towards a calculated focal point. This is all done in the Raytracer.cpp file in the Render and RenderRow functions. As mentioned in my proposal the focal point is calculated by

$$P = e + \frac{d'}{d/(d+f)}v$$

Where e is the eye position, v is the unit vector from the eye to the current pixel, d, d' are the perpendicular distance to the plane and distance from the eye to the current pixel respectively, and f is the focal length of the lens. In my code d is always one which simplified the calculation.

4.9 Final Scene

My final scene differs in a lot of ways from my proposal but I was still able to incorporate all of the objectives I listed. I wasn't able to find a good free crow model so I've placed the cat model in the scene instead. Texture mapping is used for the pillars and the water, and bump mapping is used to create the waves in the water. There is a refrective and refractive sphere in the scene, as well as a CSG model showing complex csg. CSG was also used to shape the walls and other structures in the scene. Finally I've created renders with anti-aliasing, soft shadows, depth of field, and a final scene with all of the objectives implemented. One thing I wasn't able to figure out was how to have both depth of field and anti-aliasing at the same time. For this reason my finda render doesn't have depth of field and I think it does look better without it. I tried shooting more rays for each DOF ray and averaging the values but this just seemed to get rid of the DOF effect. Regardless I did include a render of the final scene with depth of field to see what it would look like.

One of the challenges I faced was trying to find free cartoony looking textures that would fit in my scene. I couldn't find any that really worked for the walls so I had to leave them untextured. If I had more time I definitely could've created my own.

5 Additional Features

5.1 Multithreading

My multithreading implementation was fairly naive. I create a new RenderRows function in Raytracer.cpp which takes a start and end y coordinate defining a row to render. Then based on the number of threads I

split the image into that number of rows to render.

6 Bibliography

References

[1] University of Waterloo. *CS 488/688 Introduction to Computer Graphics, Course Notes, Fall 2024*. Available at: https://student.cs.uwaterloo.ca/~cs488/Fall2024/notes.pdf

[2] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation, 4th Edition*. Morgan Kaufmann, 2023. Available at: https://pbr-book.org/4ed/contents

[3] Scratchapixel. *An Online Resource for Learning Computer Graphics and Image Processing*. Available at: https://www.scratchapixel.com/index.html

[4] Jamis Buck. *The Ray Tracer Challenge: Bonus - Rendering soft shadows*. Available at: http://raytracerchallenge.com/bonus/area-light.html

[5] MIT OpenCourseWare. *Transformations in Ray Tracing*. Available at: https://dspace.mit.edu/bitstream/handle/1721.1/86191/6-837-fall-2003/contents/lecture-notes/5_1_trans_hier.pdf

[6] University of Washington. *Depth of Field for Photorealistic Ray Traced Images*. Available at: https://courses.washington.edu/css552/2016.Winter/FinalProjects/2.DOF/Final_Project_Presentation.pdf

[7] Cornell University. *Interpolated values in ray tracing, Lecture Slides from CS 4620 Computer Graphics, Fall 2019*. Available at: https://www.cs.cornell.edu/courses/cs4620/2019fa/slides/07.5rt-interp.pdf

[8] Bongjun Jin. *Selective and Adaptive Supersampling for Real-Time Ray Tracer*. Presentation from High Performance Graphics 2009. Available at: https://www.highperformancegraphics.org/previous/www_2009/presentations/jin-selective.pdf

[9] Benedict De Greve. *Reflections and Refractions in Ray Tracing*. Stanford University, 2006. Available at: https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

[10] Monument Valley 3. *Monument Valley Game Official Website*. Available at: https://www.monumentvalleygame.com/mv3

[11] CompuPhase. *Color Metrics: How to Calculate the Difference Between Two Colors*. Available at: https://www.compuphase.com/cmetric.htm