# Ray Traced Japanese Temple

Eric Haoran Huang, e48huang, 20880126

December 5th 2023

## Abstract

The goal of my project was to explore different modelling techniques and the way they can be implemented in a ray tracer, which I've done through fractal mountains, l-system trees, mesh modelling through Blender and complex constructive geometry.

## Contents

# 1 Map of the Code

The following shows the tree structure of our code.

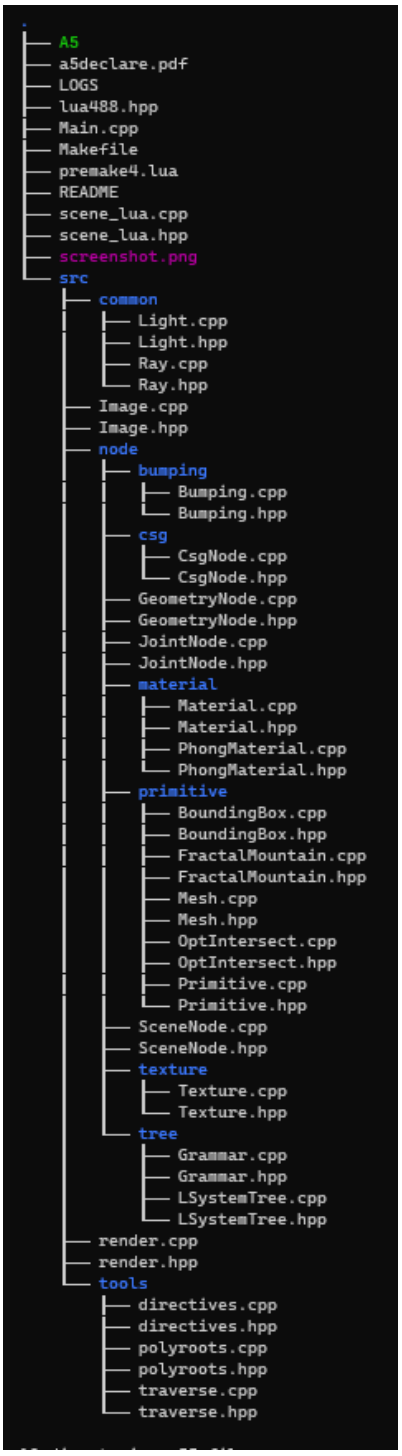



Figure 1: Running **tree** on the root folder

## 1.1 Order of Execution

The order of execution of this code goes from:

- Main.cpp gets called and interprets the Lua file by calling scene_lua.cpp.
- scene_lua.cpp builds the models we need and creates the structure we want. Mainly, it does a lot of work reading in meshes, pngs, building fractal mountains, l-system trees, and creating the CSG node structure.
- scene_lua.cpp eventually calls src/render.cpp's A4_render function.
- render.cpp's render will perform ray tracing, and go over each primitive/object's IntersectRay function multiple times, doing anti-aliasing and reflections at the same time.

Note that we require a .lua file input, an example of which can be seen under Assets/final/temple.lua.

## 1.2 Root folder organization

Root folder. This one contains documentation, Makefile generation and the entry point for the code, describing Lua commands. It also contains src/ which will contain the implementation of said Lua commands.

- Documentation. LOGS is a daily log of my work throughout the last month describing what I did and when I did it, and including some interesting tidbits hopefully. README is a document describing exactly how to use each of the new Lua commands, how I generated and modelled some models, giving credit for textures and models, and an objective list.
- Makefile generation. premake4.lua is used to change how we compile, mainly either attaching -O2 or -g for faster speeds or for being able to attach the debugger.
- Main.cpp scene_lua.(cpp/hpp) are entrypoints for the code.
- src/ folder. This folder contains all the source code needed to generate the models and the ray tracing.

## 1.3 src/ folder organization

- src/common/. Contains files that are commonly used for rendering and modelling. Light.(cpp/hpp) is used to describe lights. Ray.(cpp/hpp) is used to describe the rays we shoot in the scene and helper functions for when we need to do intersection code.
- src/node/. Will be described more in detail below, but includes all the different modelling objectives code and anything a node would have (such as Bumping and Texture) to new types of nodes such as L-System trees and CSGs.
- src/tools/. A directory used for multiple helper functions such as root finding for primitives, and directives that we can change at compile time to change the runtime of the ray tracer (such as adding print statements and toggling multithreading).
- src/render.(cpp/hpp). The actual rendering functions that will be used, including code for lighting effects such as reflections and anti-aliasing.
- src/Image.(cpp/hpp). The Image API that will save the image for us.

### 1.4 src/node/ folder organization

Generally, this file contains everything that has to deal with nodes. Either new types of nodes, or stuff that nodes need to render such as primitives and textures.

- src/node/bumping. Here is where my bump mapping is described. The bumping is an object that transforms the normal for us, which although the current implementation is static, this could be extended to provide different types of normal bumping (such as using images to provide normal bumping or some other patterns we might desire, such as non-circular waves).
- src/node/csg. This describes a new type of GeometryNode, a CsgNode which is used to define a binary operation on primitives.
- src/node/material. This here describes the material that we want. We also added in a reflectiveness double here to describe the strength of reflection.
- src/node/primitive. This here describes the different primitives and meshes that we might use. It also includes the new FractalMountain.(cpp/hpp) which subclasses Mesh. We also implemented Cylinder and Cone primitives here under Primitive.(cpp/hpp).
- src/node/texture. Includes information for adding a texture to a GeometryNode.
- src/node/tree. A new type of SceneNode that builds an L-System tree for us.

## 2 Implementation Details

### 2.1 Reflections

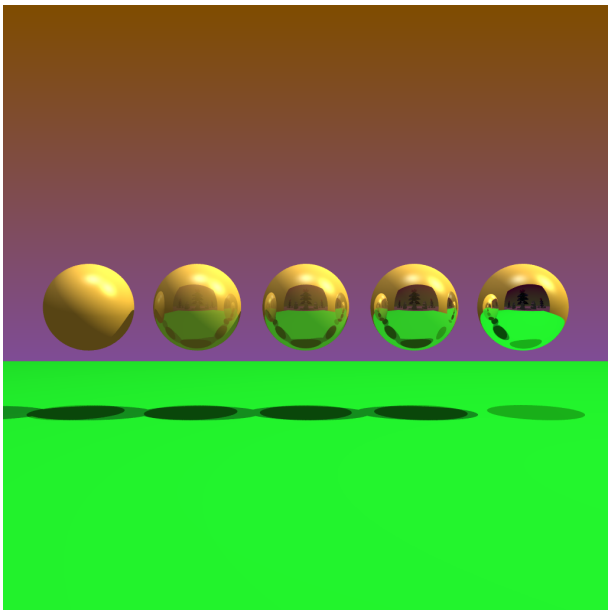


Figure 2: Reflectivity going from 0.0 to 1.0 (going right)

Reflections are implemented by making the ray tracer recursive [1]. The above is added using a new material command, gr.reflective_material() described in the README. The reflective strength parameter we pass in, measured from 0 to 1 describes how much should the material.

The equation:

$$D_{reflect} = D - 2(D \times N)N$$

given that $N$ is the normal, and $D$ is the direction of the original ray, finds the direction to recurse in to find the reflection [2]. This recursive ray tracer is found under src/render.cpp's recurse_ray_trace.

## 2.2 Texture Mapping

To add textures and any images onto any surface we want, we accomplish it through texture mapping. This ray tracer accomplishes this for meshes and cylinders, shown below. The Lua command to add a texture is <GeometryNode>:add_texture(<texture_file_name.png>).

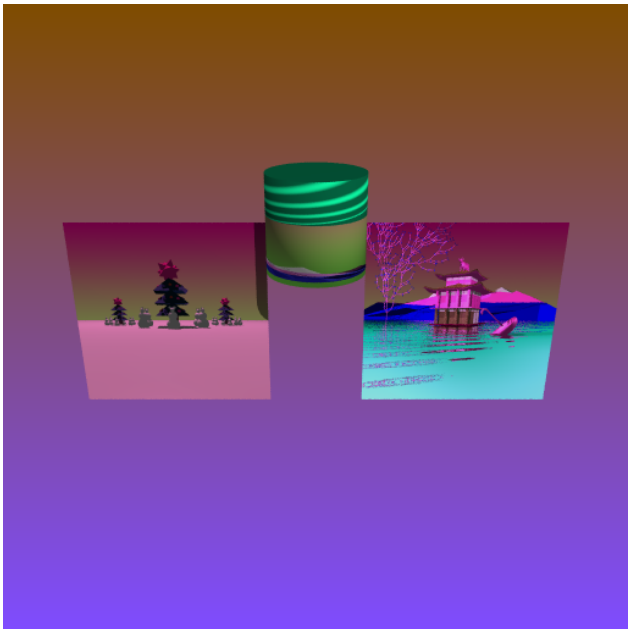


Figure 3: Texture mapped cylinder and two textured mapped plane meshes. My new art gallery.

The process for texture mapping is described below:

1. If texture mapping a mesh, edit the obj files and load them with the $(u, v)$ coordinates described in [3] and [5]. These coordinates are essentially the mapping from a vertex and the image, and we need to interpolate this coordinate depending on the ray intersection point.
2. Load in a png file, through LodePNG [4] by providing a string to the add_texture() command.
3. Based off the primitive or mesh, we need to collect these $(u, v)$ coordinates which map to a $(0, 1) \times (0, 1)$ grid.
   (a) For meshes, we need to interpolate the $(u, v)$ point by calculating the contributions from all the vertices. Luckily, this was calculated previously in A4, when needing to find whether a ray intersects a mesh at a certain face or not.
   (b) For cylinders, we need to use cylindrical coordinates to map the angle of point intersected with respect to the origin of the cylinder to a $u$ coordinate, then use the normalized height (percentage of height) of the point intersected to get the $v$ coordinate [3].

4. Map the png to these coordinates, using the LodePNG [4] library.

5. Using the $(u, v)$ coordinates, return the colour of the pixel at the png as the intersection point's colour.

## 2.3 Multithreading

To speed up the ray tracer, multithreading was used. A concurrent ray tracer is in fact simple to implement due to the simple fact that the entire screen and its calculations per pixel are independent of each other. To implement this, the thread library from C++ was used, as well as the use of mutexes.

One of the problems earlier on was discovering that depending on how you split your threads to work on the image, multithreading could essentially become single threaded without a mutex. Originally, there was just an equal horizontal split of the y's amongst the threads, however, because big mesh objects tend to concentrate around one region, this stalled quite a few renders.

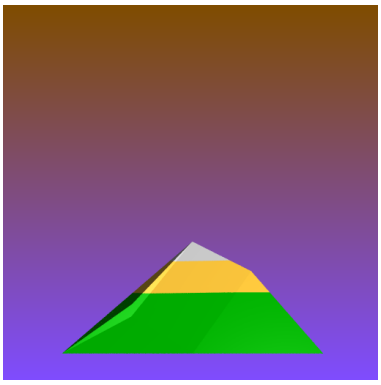
So, a mutex was introduced and threads now queue up to grab pre-defined batches based off a batch id (this changes based on size of the image) by locking the mutex and unlocking it once it had safely incremented the batch id. Using 16 threads, here are the results on the timings of the fractal mountain generation for $n$ subdivisions of faces.

| | 24 Faces Timing | 144 Faces Timing | 864 Faces Timing |
|---|---|---|---|
| Multithreading | 0.4573s | 2.0983s | 12.2149s |
| No Multithreading | 1.6860s | 7.9621s | 47.1330s |

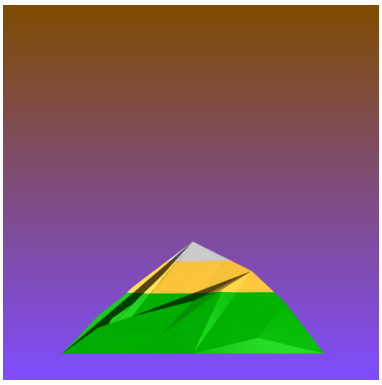
There is a clear four times speed improvement. As the number of faces increase as well, there is also the same number of time jumps as well, around six times for both multithreading and non-multithreaded. However, do note that in this case here, the 16 threads were not exactly useful as it seems as though only four threads at once actually were used. A future extension could be to automatically change the number of threads depending on the number of faces and the concentration of the objects.

## 2.4 Fractal Mountains

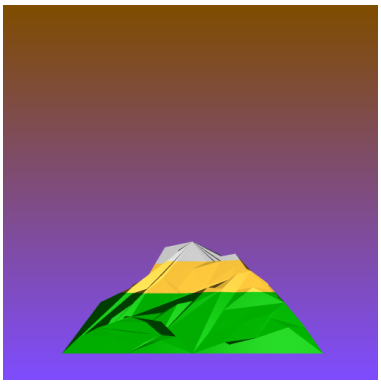
Fractal mountains are added to the scene by doing gr.fractal_mountain() according to the README specification. In it, we must include the starting dimensions, a measure of how rough the mountain is, and the number of subdivisions we do (how many times we will try and split the faces to describe more detail).



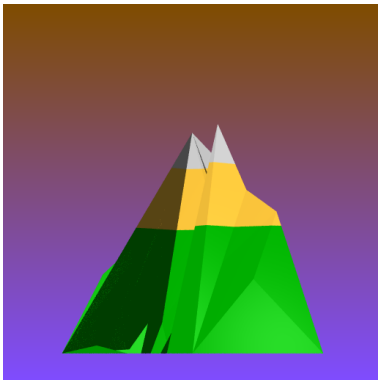
Roughness variance of 0.1 and 1 subdivision of faces



Roughness variance of 0.1 and 2 subdivisions of faces



Roughness variance of 0.1 and 3 subdivisions of faces



Roughness variance of 0.5 and 2 subdivisions of faces

Figure 4: Mountains of varying variance and subdivisions of faces with the same starting dimensions

It's clear that as the variance increases, the triangle faces tend to be sharper and longer, and create taller peaks. As the number of subdivisions of faces increase, the larger the perceived detail increases.

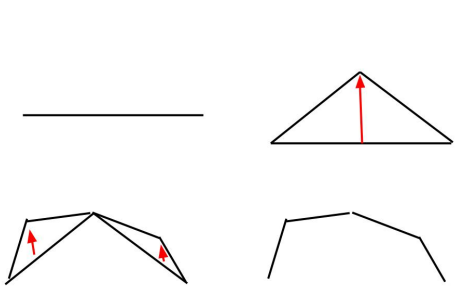
Fractal mountains were done by doing repeated subdivisions over a triangle face's line segments and perturbing them randomly in the y direction [6]. This perturbation is described by the following:

$$\{(x_1, y_1), (x_2, y_2)\} \implies \{(x_1, y_1), (\frac{x_1 + x_2}{2}, f(\frac{y_1 + y_2}{2}, \frac{x_1 + x_2}{2})), (x_2, y_2)\}$$
$$f(y, \delta x) = y + (\Delta x)N(1, \sigma^2)$$

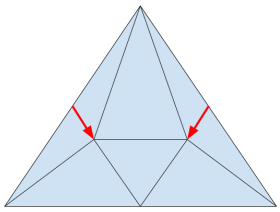
where we take the midpoint of each line segment and randomly add noise to the y direction using a normal distribution. Note that the above is generalized to the 3D case for the fractal mountains.

The following figure shows how we do this random walk for a single line segment, as well as how we split the faces for the 3D case, for the fractal mountains. The faces were split by perturbing two line segments

up in the y direction, and then generating 6 faces from it.



Stochastic process of Random Perturbation



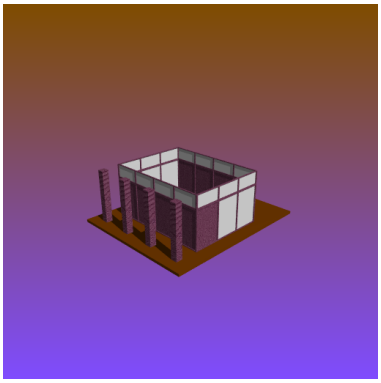
Fractal Mountain Face 6-face Split

Figure 5: Mountains of varying variance and subdivisions of faces with the same starting dimensions

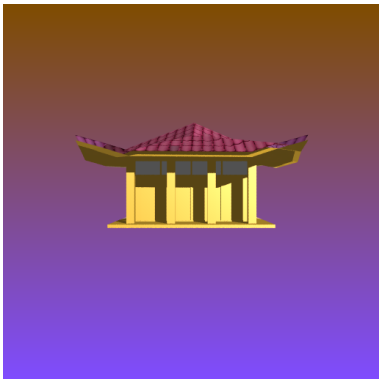
The starting shape (note that a future extension would be to provide any shape the user wants as a mesh input) is a rectangle based pyramid. This is done such that there is one peak, and that this peak will stay at the center as we perturb the rest. There were four faces that connect to the peak which is then perturbed, with each of those 6 perturbed faces being perturbed further and further until the subdivision depth is reached.

## 2.5 Temple Modelling

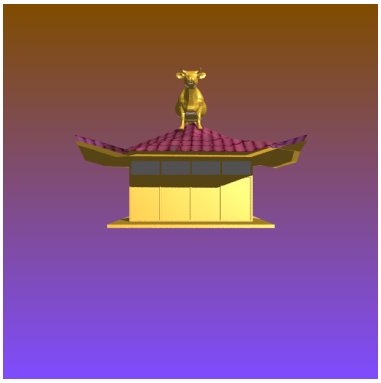
The temple modelling was learning how to do procedural generation through Lua, and creating a complex model. The following are pictures of how I built the temple bit by bit.



First Floor



Second Floor



Third Floor

Temple

Figure 6: Modelling of different parts of the Temple

The following is a snippet of what the procedural generation used was, in particular to build the pillars on the first floor. This type of for loop generation was reused throughout the generation of this temple.

```lua
-- now we want to add in the big pillars
-- pillar section
local pillar = gr.mesh('pillar', 'cube.obj')
pillar:set_material(grass)
pillar:scale(0.25, 2, 0.25)
pillar:add_texture('lighter_wood.png')

local pillar_struct = gr.node('pillar_struct')
for i = 1, 4 do
    tmp_pillar = gr.node('pillars' .. tostring(j))
    tmp_pillar:translate((i - 1) * 2.5, 0, 0)
    tmp_pillar:add_child(pillar)
    pillar_struct:add_child(tmp_pillar)
end
pillar_struct:translate(-3.75, 0, 5)
first_floor:add_child(pillar_struct)
```

Figure 7: Modelling of the pillars used on the first floor

## 2.6 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is used to construct and model solid geometry through the use of three boolean operations of union, difference and intersect. To accomplish these three boolean operations once isn't too difficult, as described in [1], all that's needed is to:

1. Grab the segments of the primitive that's being intersected. For this ray tracer, all of them had at most one segment. Define this as a SegmentIntersect() function.
2. Using these segments, apply the boolean operations to two lists of these segments.
   (a) For simple CSG, this involves applying the boolean operation to two lists of size one.
   (b) For complex CSG, this involves applying the boolean operation to two lists of arbitrary size.

   Return the result as a SegmentIntersect() function.
3. Just as the case for intersecting points, simply return the closest point taken from the result of a SegmentIntersect().

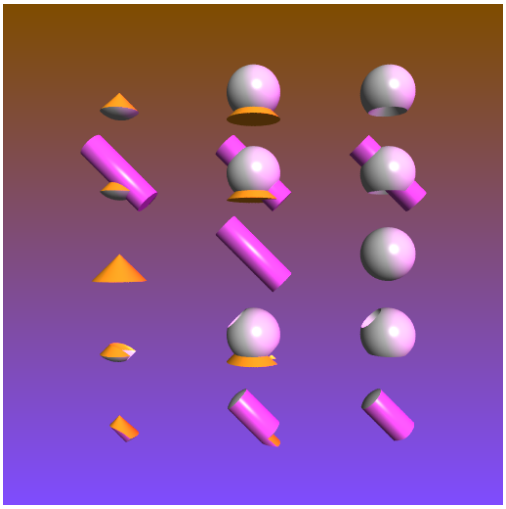
To make the code easier to deal with, new Segment and SegmentList data structures were created to store these sections. To do simple CSG, this was not tricky at all, just implement the following functions (given that $A = (a_1, a_2), B = (b_1, b_2)$):

1. Segment Difference, returns a segment list of the result of A - B. There were four cases in this part here (given that they intersect).
   (a) If B completely covers A, return nothing.
   (b) If A completely covers B, then return two segments of $(a_1, b_1), (b_2, a_2)$.
   (c) If A starts before B and doesn't completely cover, then return a single segment of $(a_1, b_1)$.
   (d) If A starts after B and doesn't completely cover, then return a single segment of $(b_2, a_2)$.

   As described in [1], there is a need to flip the normal on difference as we want to show that the inside is in fact solid.
2. Segment Union. Given that they intersect, simply return $(\min(a_1, b_1), \max(a_2, b_2))$.
3. Segment Intersect. Given that they intersect, return the segment created from the 2nd and 3rd closest points.

Using the above functions, we can also implement complex CSG. Complex CSG is simply an extension of the above, except that we need to carefully iterate over lists. To help with this, an invariant is imposed, such that every list that is returned from a SegmentIntersect() must be disjoint and in increasing order.

Having this, we can then simply carefully iterate over two lists, and doing the above segment difference, unions and intersects when necessary (i.e. when there is an intersection). Careful consideration needs to be applied to difference especially in this case since a single segment in A can have multiple segments in B remove it. The result of this is as shown below.



Different CSGs

Umbrella Model

Figure 8: Complex CSG in Action

The complex CSG model has the following CSG:

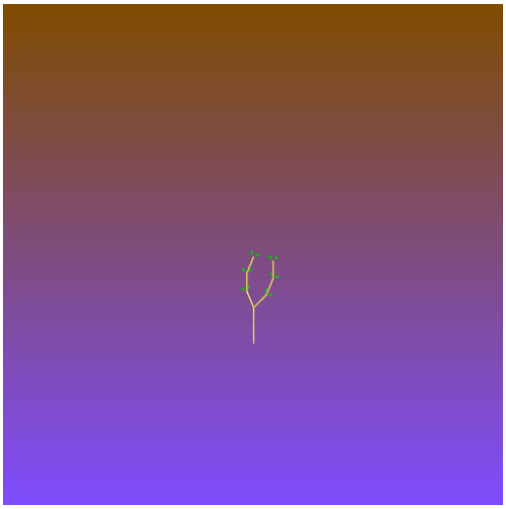
1. Top row: combining cone with sphere in different ways (intersection, union, difference).
2. Second row: Combined first row with long cylinder primitive through union.
3. Third row: Showing off primitives.
4. Fourth row: First row minus cylinder.
5. Fifth row: First row intersect cylinder.

The umbrella model has the following CSG:

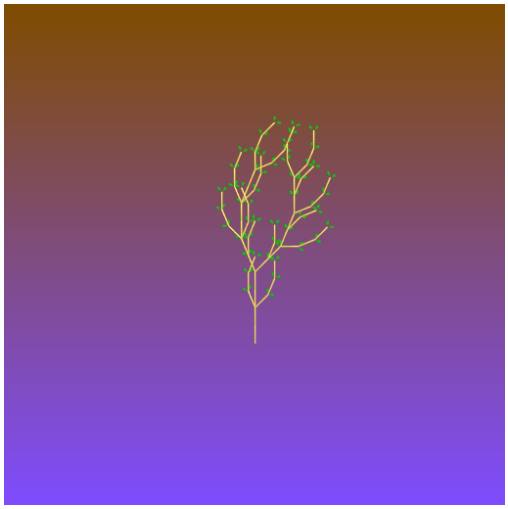
1. Top part: Sphere hollowed out another sphere.
2. Handle: A long cylinder (arm) unioned a sphere that was cut in half by a cube that was then cut out with a smaller sphere.

## 2.7 L-System Trees

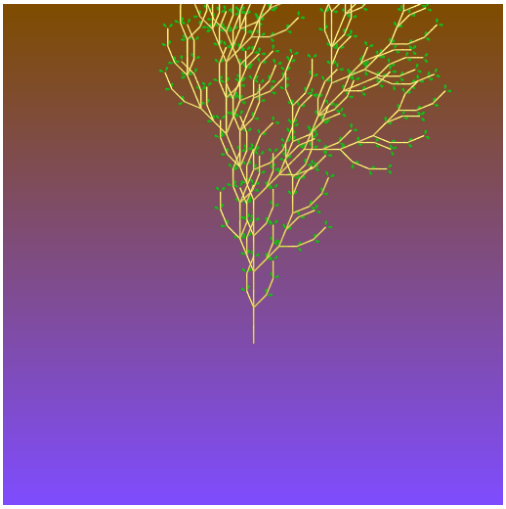
L-System trees are a method of taking context-free grammars, expanding on them and using the result as an encoding of what we want a plant to look like. Using the gr.tree() command (more details in README), the ray tracer takes a grammar file, number of iterations and generates a new LSystemTree node that acts as a SceneNode. The following is the result of this process, of iterating on the grammar shown below at different amounts of substitution depth.



Tree with Substitution Depth 1



Tree with Substitution Depth 2



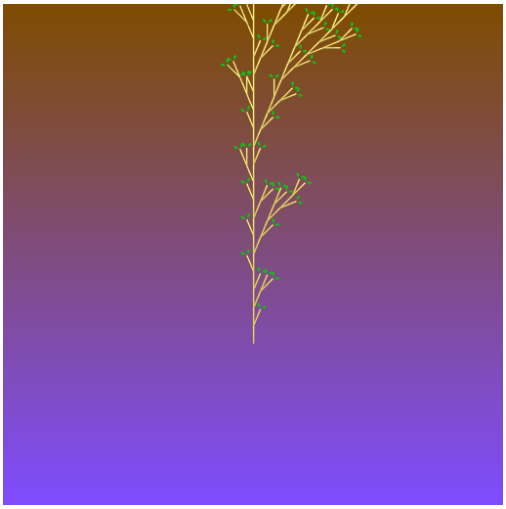
Tree with Substitution Depth 3

```
F

trunk F 0.1 1.0 0.1
leaf leaves 0.25 0.25 0.25
branch ( ) 0 0 -22.5
branch [ ] 0 0 22.5
branch { } 0 0 45
branch < > 0 0 -45

F 1.0 F F (
    ( F { leaves } < leaves > [ F { leaves } < leaves >
        [ F { leaves } < leaves > ] ] )
    [ [ F { leaves } < leaves >
        ( F { leaves } < leaves > ( F { leaves } < leaves > ) ) ] ] )
leaves 0.5 leaves
leaves 0.15 { leaves } leaves
leaves 0.15 < leaves > leaves
```

Tree .grammar File

Figure 9: L-Systems Tree Demonstration, grammar inspired from [7]

Above, it's clear that the tree is self similar, which isn't exactly how nature acts. By using stochastic processes we can get rid of this problem, as shown below. Note how the stochastic process doesn't repeat the smaller sections in the larger sections. In the ray tracer, this is accomplished by adding a probability next to each rule such that it expands at that probability.



Flower with Stochastic Branching

```
F

trunk F 0.1 1 0.1
leaf leaves 0.25 0.25 0.25
branch ( ) 0 0 -22.5
branch [ ] 0 0 22.5
branch { } 0 0 -45
branch < > 0 0 45

F 0.33 F ( F { leaves } < leaves > ) F [ F { leaves } < leaves > ] F
F 0.33 F ( F { leaves } < leaves > ) F
F 0.34 F [ F { leaves } < leaves > ] F
```

Flower .grammar File

Figure 10: Stochastic L-System Trees Demonstration, grammar inspired from [7]

What makes this ray tracer special with L-System Trees is the ability to generalize to any grammar we provide it, as well as the ability to change what the trunk and leaves are interpreted as, and branching angles all written through the '.grammar' file. It is accomplished through the following steps.

1. Parses in a tree grammar. See README or the next subsection for details on the grammar itself.
   (a) Parses in the rules and starting symbol to expand on the grammar later.
   (b) Parses in the geometric interpretation for a symbol. This can either be a trunk, a leaf or a branch. For trunks and leaves, this will turn into a GeometryNode specified by the Lua command. For branches, this is specified within the grammar file itself and will mark a rotation branching from the previous node. Note that what's important to making L-System Trees work is to provide a geometric interpretation for the symbols [7], to actually generate a model we can look at.
2. Does a stochastic approach depending on the grammar specified to expand on the grammar and get a resulting string. This is accomplished through parsing in the rules in step 1(a), and doing string substitution according to the probability distribution defined.
3. Takes the resulting string and parses into an Abstract Syntax Tree (AST). The general parsing method is that if it's surrounded by brackets, then everything within the brackets is a subtree. That subtree's parent is the previous non-bracketed symbol. If it's a non-bracketed symbol, then its parent is the previous non-bracketed symbol as well.

   This means that $A_1(A_2)[A_4]A_5(A_6)$ has that $A_2, A_4$ branch off of $A_1$ but $A_5$ continues straight from $A_1$ and $A_6$ branches off $A_5$.
4. Takes the AST and generates a subtree of GeometryNode's and SceneNode's.

### 2.7.1 Grammar File Specification

For L-system trees, we need to be able to define a Context-Free Grammar (CFG) that models the tree. The following describes how we can generate and describe these trees.

1. <start-symbol>

2. trunk <symbol> <x-scale> <y-scale> <z-scale>

3. leaf <symbol> <x-scale> <y-scale> <z-scale>

4. branch <left-bracket> <right-bracket> <x-rotate> <y-rotate> <z-rotate>

5. <symbol> <probability> <list-of-symbols>

Here, we define a grammar such that 1. denotes the starting symbol upon which the grammar will expand from. Then 2. and 3. mark how we will interpret each of the symbols, either as a leaf or a trunk defined from the gr.tree command. These can be any arbitrary geometry node. Then, the scaling tells us how to change the size of these nodes. This is useful for creating trees that differ in size depending on where in the tree it is (e.g. trunks are smaller than branches).

4. marks the different type of branches we can do. It marks what symbols will be read as a left bracket and right bracket that marks which symbols belong to a branch. For example, F ( F F ) marks that we will branch after an initial F, to have two F's conjoined. It also marks how we rotate when we branch, described under x-rotate, y-rotate, z-rotate. Note that we rotate around the axes in order, i.e. we do x-rotate amount, then y then z.

5. then marks the expansion rule we apply. This will tell us how we want to expand on the grammar given a probability. For example,

(a) F 0.3 F F

(b) F 0.5 F ( F F ) F

(c) F 0.2 [ F F ] F

F has a 30% chance of expanding to F F, 50% to F ( F F ) F, and a 20% chance to expand to [ F F ] F.

Note: Since we are using spaces as our delimeters, we have to write our rules as ( F F ) and not (FF) or (F F). This allows us to have strings as symbols, not just characters. And an unlimited amount of brackets (can use iamleftbracket iamrightbracket as brackets).

Note that for 2-4 we can have as many of these as long as we don't have conflicting symbols used. For 5., we can have as many rules as we want, though the probabilities need to add to 1.0 given a symbol. If not, a hidden rule of disappearing is added.

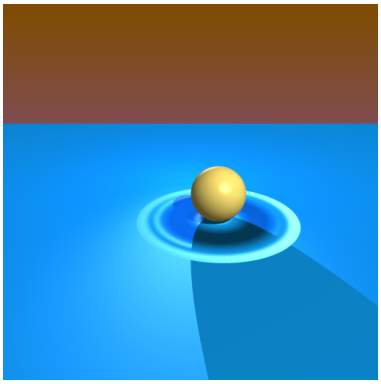
## 2.8 Normal Mapping

To add the wave effects, the normal needed to be perturbed. For this to be well-defined, the command was <GeometryNode>:mesh_add_bumps((u, v), num-periods, period), more specifically, a GeometryNode that was a mesh, as the plan was always to use normal mapping onto meshes. The $(u, v)$ would define the coordinates actually specified by the texture map (see Section 2.2) which would be the starting point of a circular wave. The period defines how big the sections of the normal perturbation are and the number of periods defines how long the waves last for. This circular wave, was defined very simply by the following (given the $(u_i, v_i)$ coordinate of the intersection point):

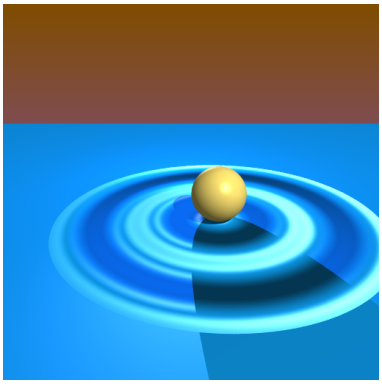
$$\theta = \cos(\frac{2\pi d}{period})$$
$$d = ||(u, v) - (u_i, v_i)||_2$$

where $\theta$ represents how much to perturb the normal by in order to create the illusion of a bumpy surface [1]. The $\theta$ was then used to rotate about the x-axis. This works because the angle $\theta$ changes periodically

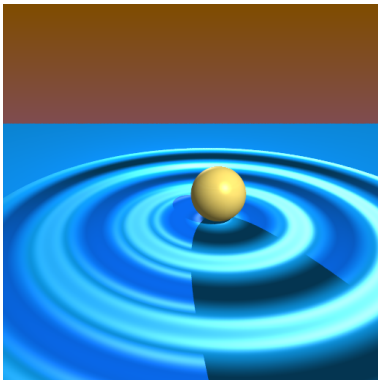
(in fact at every period that is user defined) relative to the distance away (described by $d$) resulting in a normal that gets perturbed such that it points towards the positive z direction and away from the positive z direction. This leads to the interactions shown below.



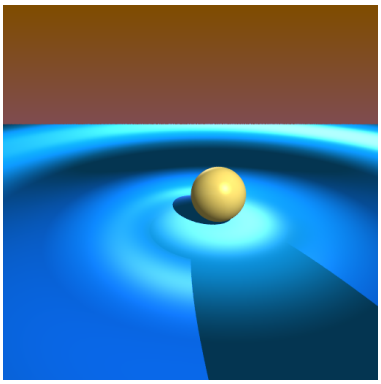
Period = 0.1, Num. of periods = 1



Period = 0.1, Num. of periods = 2



Period = 0.1, Num. of periods = 3



Period = 0.5, Num. of periods = 2

Figure 11: Effect of Normal Bumping with different periods and limit of waves

The desired effect is achieved, as the period increases, the distance between light and dark also increases, and as the number of periods increase, the stopping of the waves also gets further away from the centre.

## 2.9 Extra Objective: Extra Primitives

To do some of the CSGs as well as do some of the necessary L-System trees (cylinders are used as trunks), cylinders and cones were implemented as extra primitives. Their intersection equations were the same as were found in [1].

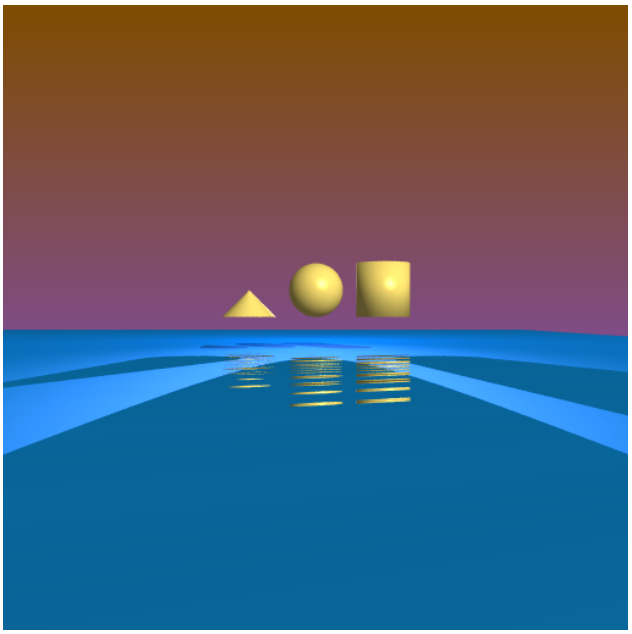
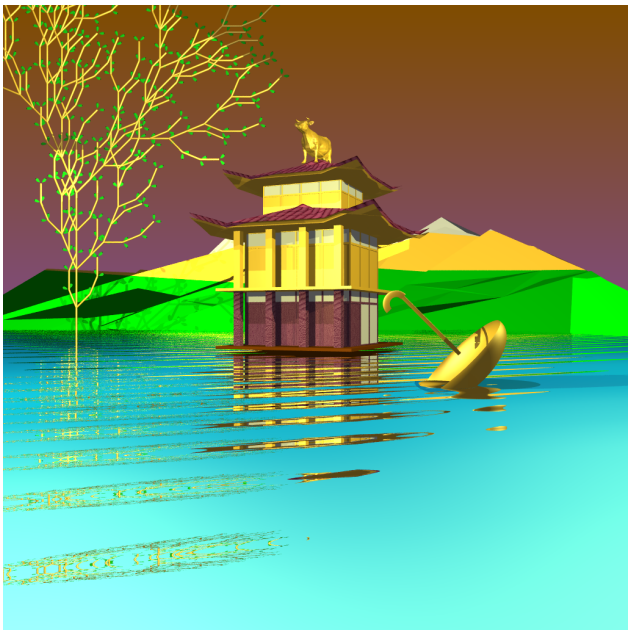


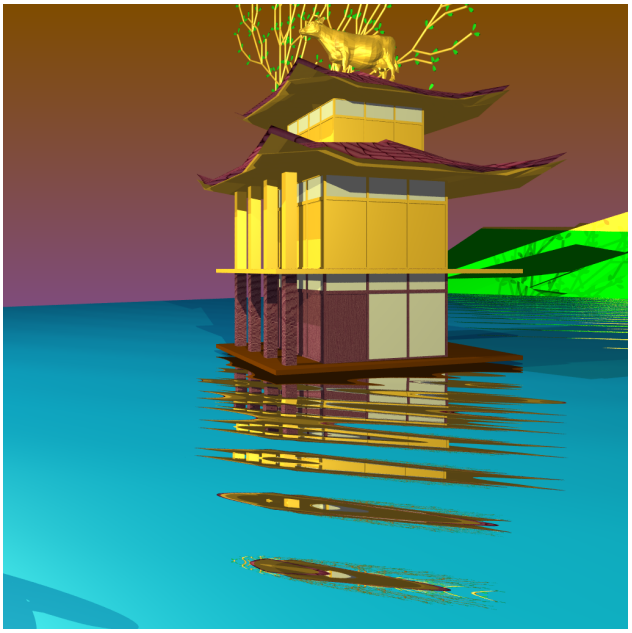
Figure 12: Cone, Sphere and Cylinder Reflecting on the Meaning of Life

## 2.10 Final Scene

Putting everything together...



(a) Final front view



(b) Side view

Figure 13: Final Figure!

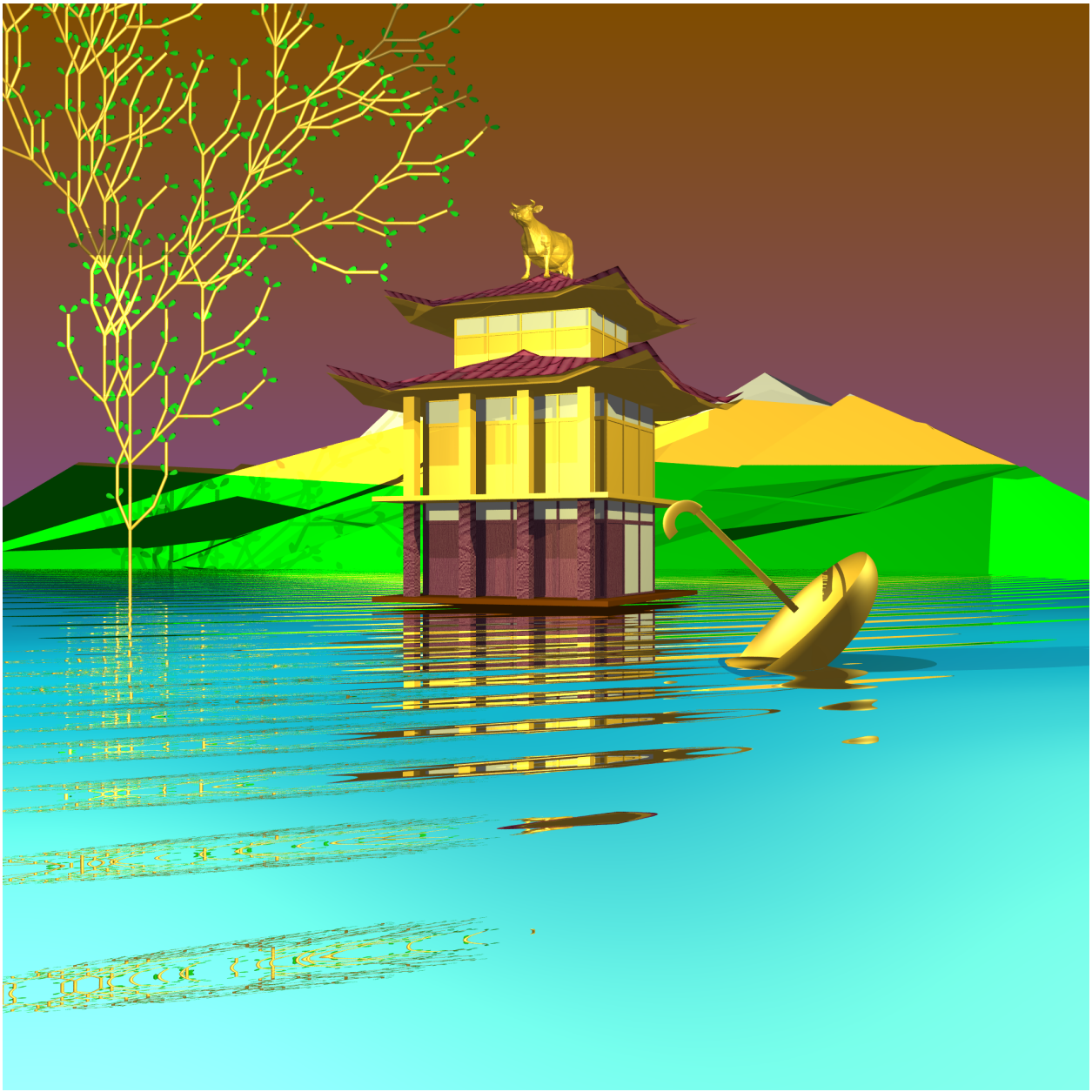


Figure 14: Bigger Resolution Final Pic

# 3 Bibliography


1 **CS 488/688 Fall 2023 Course Notes**, Mann, University of Waterloo, 2023, pg. 138-139, 142-144, 145, 193, 194-201.

2 **Recursive Ray Tracing**, Stanford CS 148 Introduction to Computer Graphics and Imaging, 2023 pg. 8-18.

3 **Texture Mapping**, The Ray Tracer Challenge.

4 **LodePNG**, LodePNG.

5 **Wavefront Obj File**, Wikipedia, 2023.

6 **Introduction to Computer Graphics**, Foley, Van Dem, Feiner, Hughes, and Phillips, 1994, pg. 362

7 **The Algorithmic Beauty of Plants**, Prusinkiewicz, and Lindenmayer, 2004, pg.21-27.


# 4 Acknowledgements


I'd like to acknowledge the wonderful time I had during this course! What a ride this was!

Seriously though, I'd like to thank everyone for being so encouraging and kind during this time here. I'd like to thank everyone at the render rave too (what a time that was!), Adam, Zack (especially for organizing) and someone's name I honestly forgot and am too scared to ask. And of course my buddy Michael who's been through my panic bouts in the graphics lab.

Finally, but definitely not last, big thanks to the TAs and Prof. Stephen Mann for the great semester!

# 5 Objectives

**Full UserID:_____ Student ID: e48huang, 20880126**

___ 1: Objective one. Easy goal: Reflections in the water of the temple.

___ 2: Objective two. Easy goal: Texture Mapping for the different paneling shown on the temple.

___ 3: Objective three. Easy goal: Multithreading to speed-up ray tracing.

___ 4: Objective four. Medium goal: Fractal mountains for the background

___ 5: Objective five. Medium goal: Do procedural generation of the temple modelling through Lua file generation or using a context free grammar and add texture mapping.

___ 6: Objective six. Medium goal: Simple CSG - two objects only.

___ 7: Objective seven. Medium goal: Trees through L-system plants.

___ 8: Objective eight. Medium goal: Normal Mapping in the water to create a wave effect.

___ 9: Objective nine. Hard goal: Complicated CSG for an upside down umbrella in the water.

___ 10: Objective ten. Final goal (medium): Finish the final scene with everything (without the extra objectives).

Extra goals if time permits. Note: I do not expect to get all of these done, rather probably 1 or 2 of them. This is more so as ideas on how I can continue working on making this even better.

1. Edit: Instead of having this as an objective, I moved it to the extras. A skybox to simulate a larger landscape.
2. Take into account atmospheric scattering and have the atmosphere show different colours at different times of day. This will be good to showcase a good night scene.
3. Extend the reflections and do an underwater scene looking up - use Snell's law for refraction and internal reflection to get a more accurate description of how water works. I would provide an underwater scene for this.
4. Hierarchical scenes speed-up through Bounded Volume Hierarchies and a night scene.
5. Roof paneling through CSG and torus modelling.