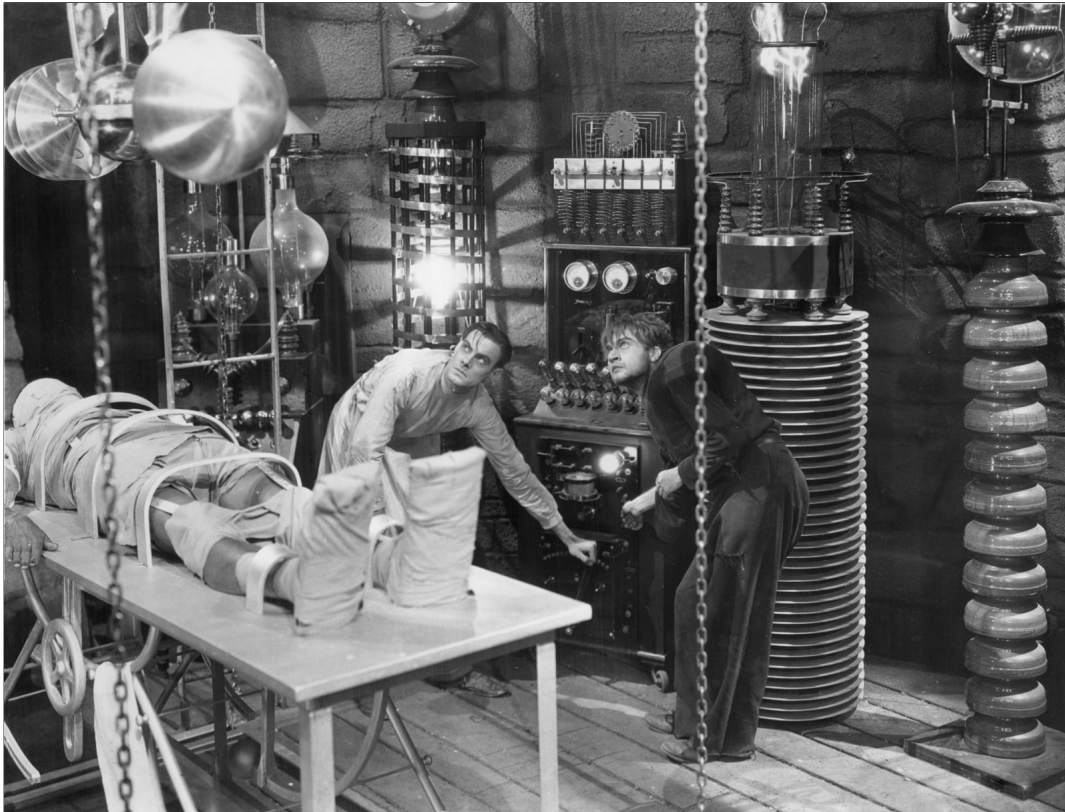


1 Introduction

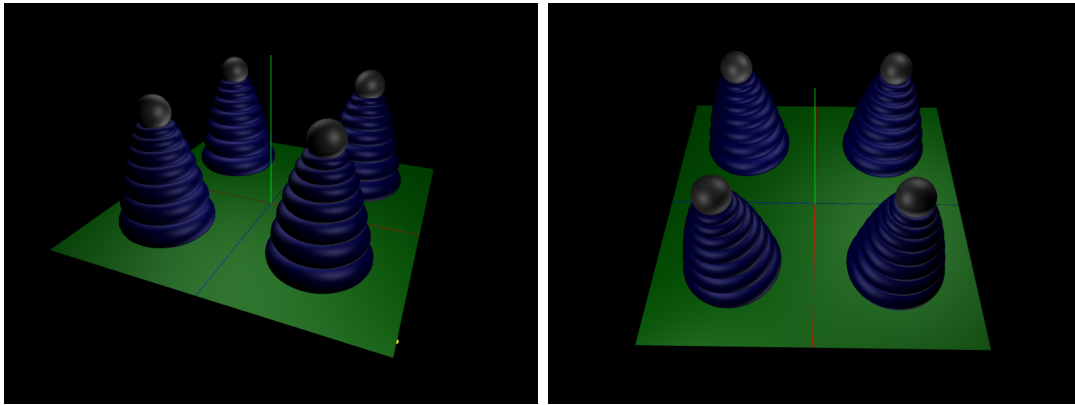
As usual, zip the project directories into one zip file. Upload the zip file to the Homework #6 page of the MyClasses site for this class. These exercises are extensions of the multiple point lights example. Each exercise should have three light sources with one that is movable with the UI, as in the multiple point lights example.

This exercise set was inspired by the old Frankenstein films with all the electronic fireworks on the set of the laboratory.



2 Exercise #1

This exercise is to create 4 towers on a plane. The plane is on the xz -plane and is a dull green material. The towers are a sequence of 7 tori that are stacked on top of each other and getting smaller as they go up. The material for the tori is a dull blue. The top is a sphere with material a dull dark grey. Note that none of these colors are predefined.



The UI should have all the options as in the multiple point lights example. Specifically,

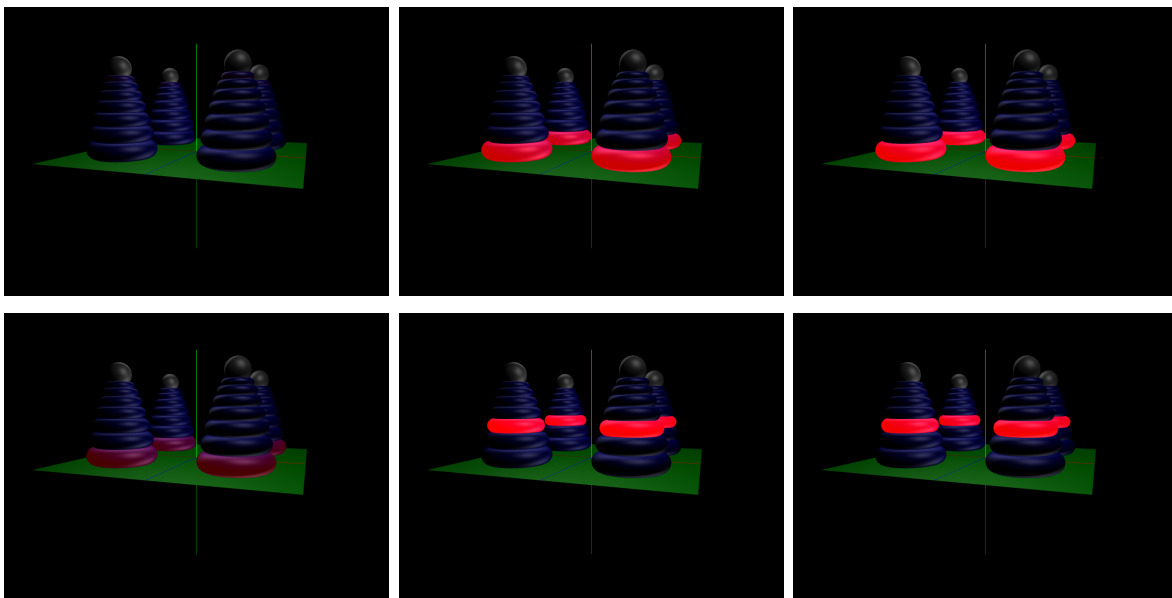
- User Options — Keys
 - Escape: Ends the program.
 - C: Toggles between the two cameras.
 - O: Toggles between outline and fill mode for the box and cube objects.
 - L: Toggles the drawing of the axes.
 - K: Toggles the drawing of the light position.
 - F1: Draws in fill mode.
 - F2: Draws in line mode.
 - F3: Draws in point mode.
 - F4: Toggles between 60 FPS and unlimited FPS.
 - F12: Saves a screen shot of the graphics window to a png file.
- If the spherical camera is currently selected,
 - If no modifier keys are pressed:
 - * Left: Increases the camera's theta value.
 - * Right: Decreases the camera's theta value.
 - * Up: Increases the camera's psi value.
 - * Down: Decreases the camera's psi value.

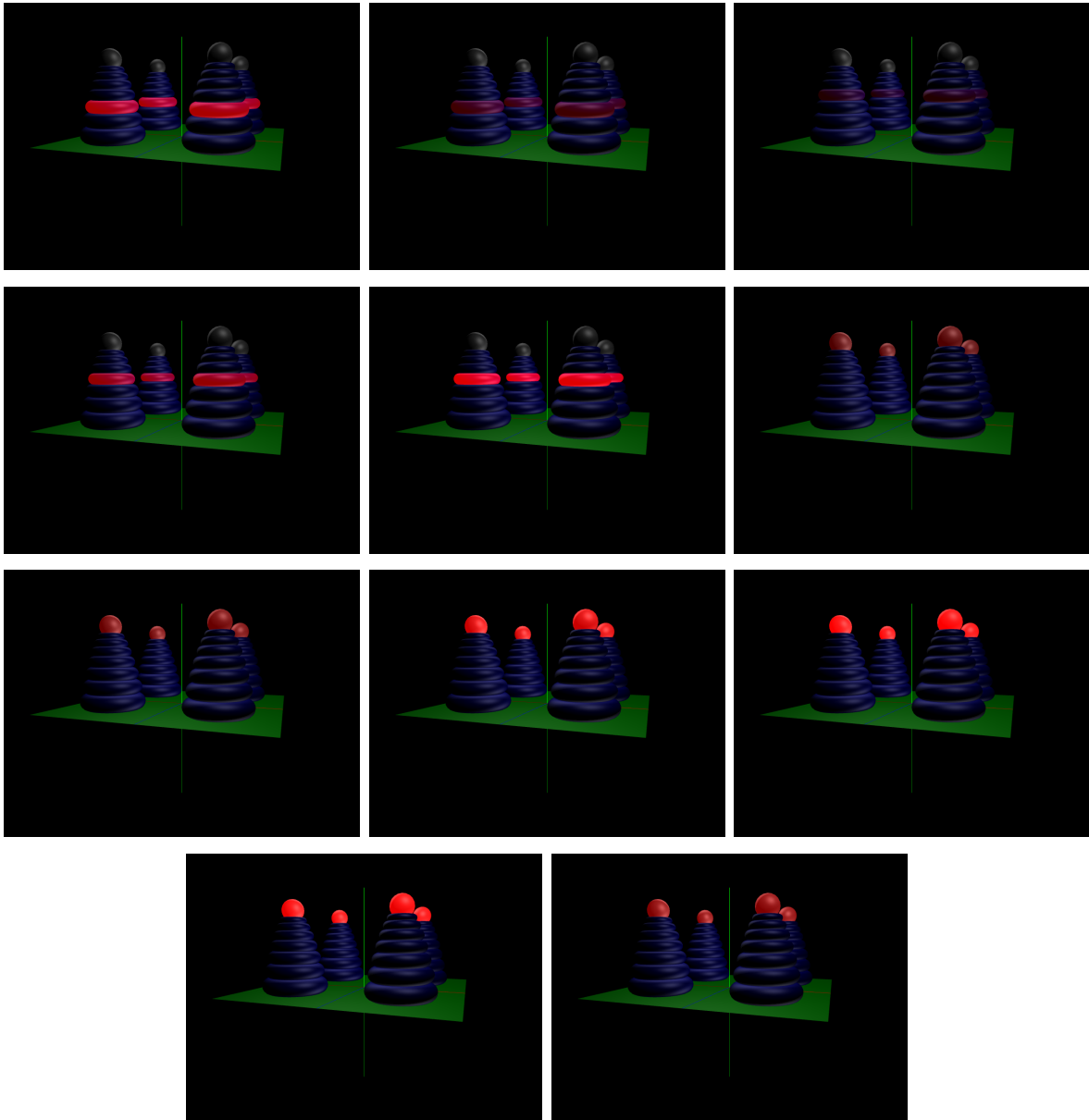
- If the control or Z key is down:
 - * Up: Decreases the camera's radius.
 - * Down: Increases the camera's radius.
- If the yaw-pitch-roll camera is currently selected,
 - If no modifier keys are pressed:
 - * Left: Increases the yaw.
 - * Right: Decreases the yaw.
 - * Up: Increases the pitch.
 - * Down: Decreases the pitch.
 - If the control or Z key is down:
 - * Left: Increases the roll.
 - * Right: Decreases the roll.
 - * Up: Moves the camera forward.
 - * Down: Moves the camera backward.
 - If the shift or S key is down:
 - * Left: Moves the camera left.
 - * Right: Moves the camera right.
 - * Up: Moves the camera up.
 - * Down: Moves the camera down.
- If the alt or X key is pressed the spherical camera that is attached to the light is altered:
 - Left: Increases the light's theta value.
 - Right: Decreases the light's theta value.
 - Up: Increases the light's psi value.
 - Down: Decreases the light's psi value.
 - (control or Z) + Up: Decreases the light's radius.
 - (control or Z) + Down: Increases the light's radius.
- User Options — Mouse
 - If the spherical camera is currently selected,
 - * If no modifier keys are pressed and the left mouse button is down a movement will alter the theta and psi angles of the spherical camera to give the impression of the mouse grabbing and moving the coordinate system.
 - * If the control key is down and the left mouse button is down then the camera will be moved in and out from the origin by the vertical movement of the mouse.

- * If the wheel is moved then the camera will be moved in and out from the origin by the amount of the wheel movement.
- If the yaw-pitch-roll camera is currently selected,
 - * If no modifier keys are pressed and the left mouse button is down a movement will alter the yaw and pitch angles of the camera.
 - * If the control key is down and the left mouse button is down then the camera will be moved forward and backward by the vertical movement of the mouse.
 - * If the shift key is down and the left mouse button is down then the camera will be moved right and left as well as up and down.
 - * If the shift and control keys are down and the left mouse button is down then the camera will roll.
 - * If the wheel is moved then the camera will be moved forward and backward by the amount of the wheel movement.

3 Exercise #2

This exercise will add onto the last one. We will add in some materials animation. This program will make each of the tori and then the spheres glow red. The red will fade in and fade out on the bottom level of tori then the red glow moves up to the second level tori, then the third, and so on up to the spheres. Then the process will repeat. Each fade in and fade out will take exactly one second on each level. The speed should be the same no matter what the FPS is. The user interface should be the same as with the previous exercise.





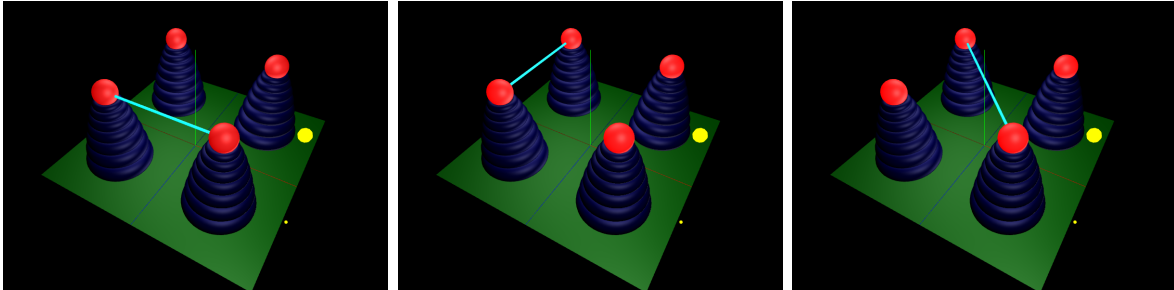
4 Exercise #3

This exercise will add onto the last one. In this exercise when the top spheres are doing their glow fade in and out there are random lasers between the spheres. On each frame two spheres are chosen at random and for that frame a laser beam will be between those two spheres.

To make this happen create a cylinder class that creates a cylinder with one end at the origin and it moves along one of the coordinate axes. Have the class take as parameters the radius of the cylinder and the length. Also have as parameters the number of divisions in each direction as well a beginning and ending angles to the curved portion of the cylinder,

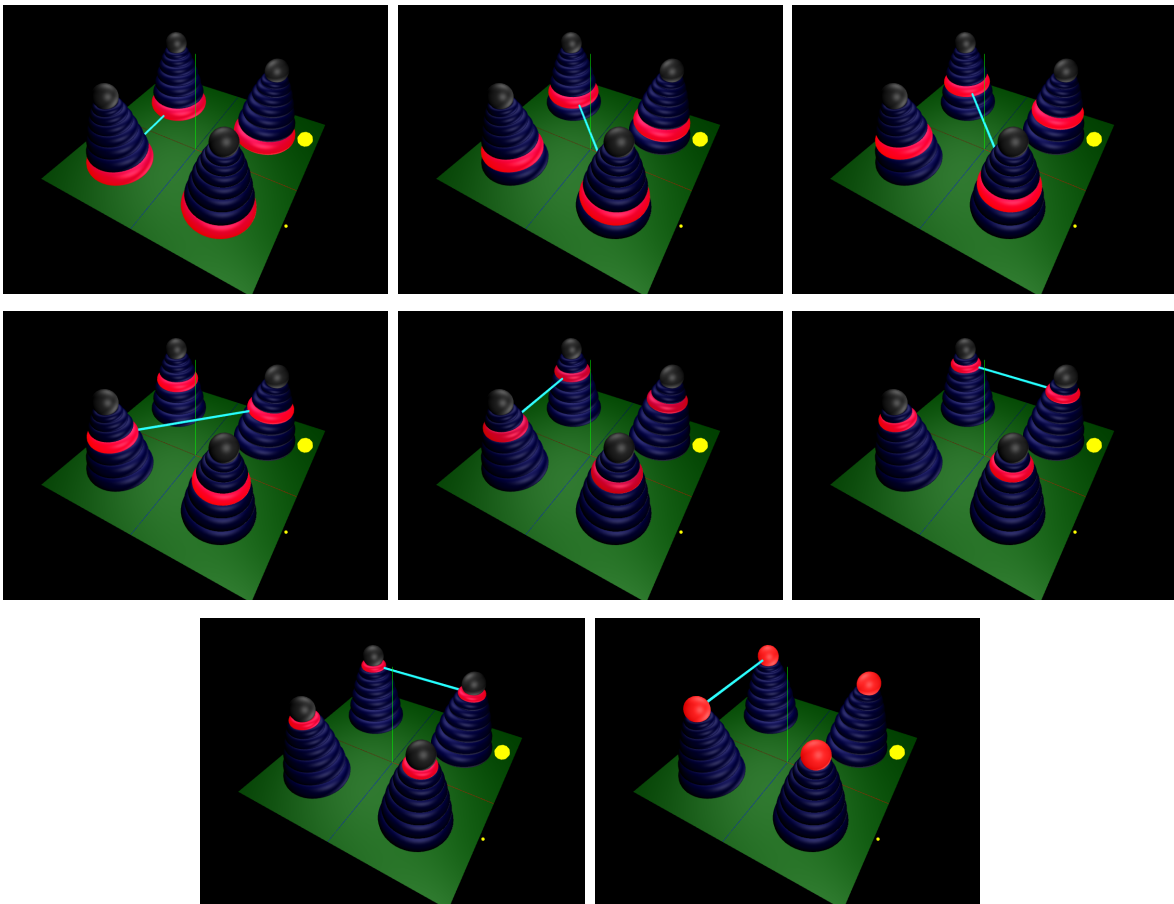
this will allow the construction of a partial cylinder, like a trough.

Load only one cylinder into the graphics engine and use transformations to move and position the laser beam. The user interface should be the same as with the previous exercises.



5 Exercise #4

This exercise will add onto the last one. With this exercise the laser beams are produced at each level as long as the level is in its fade in and fade out glow stage. The beams should be at a height in the middle of the stage that is glowing. The user interface should be the same as with the previous exercises.



6 Challenge Exercise: Optional for Extra Credit

Read the handout *Visual Simulation and Rendering of Lightning* by Samuel Atkinson and Ian Chamberlain which is at the end of this document. Concentrate on the first three section on the creation of the model. The sections on rendering will not apply to us since they are implementing the model in a ray tracer and not a real-time system like what we are working with.

Our goal is to create a spark-like animation between two spheres. We will do this in three stages. First we will develop a simple single branch lightning, the “Downward Leader”. Then we will add in branching and finally put it into an on/off animation. The user interface should be the same as with the previous exercises.

6.1 Exercise #5a

Create a spark class that takes as parameters the starting point, ending point, color, maximum segment angle, and the mean segment length. Use their algorithm to generate a series of line segments from the beginning point to the end point. These are all the parameters we will need at this point, when we include the branching we will add in the other options. Since this is simply a sequence of line segments it is better to draw it using the same shader as you use for the axes.

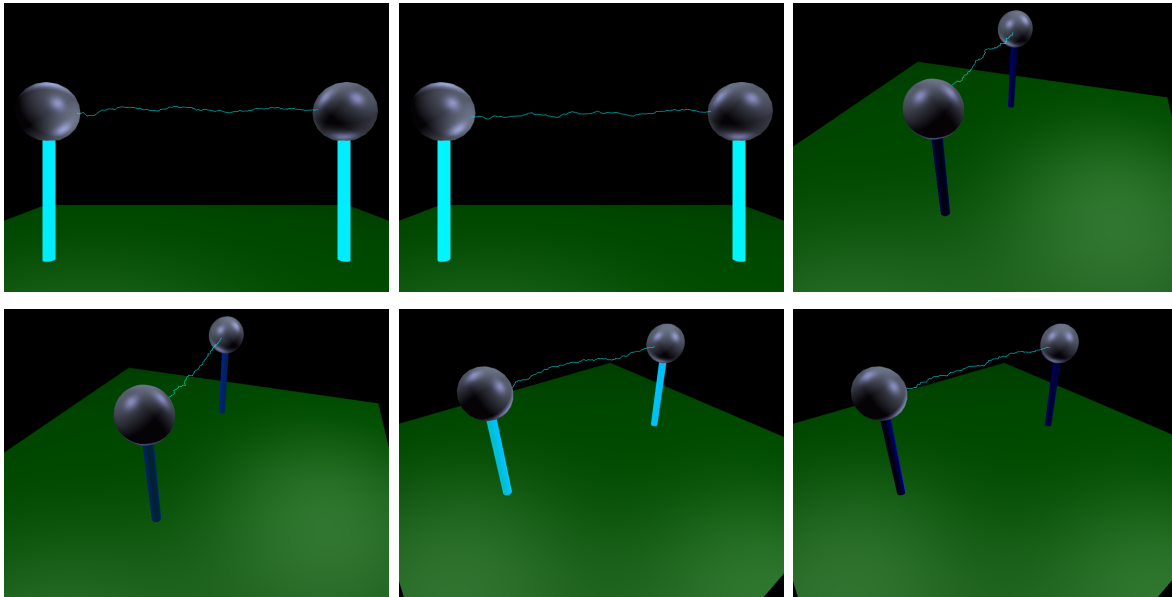
In the algorithm I made a couple changes. The algorithm is essentially to model lightning, the spark is lightning with a target. While their algorithm does get close to a target it is not set up to hit the target directly. In fact, the larger the segment angle and the longer the trek the more variability is introduced and the further away from the target the bolt ends. Here are the alterations that should be made.

- The rotation normal vector is set randomly at each segment, instead of being fixed.
- To make the target hit, instead of fixing the direction as the original direction from the starting point to the ending point, we reset the direction at each segment to be the direction from the current position to the ending point. If we were modeling lightning it would be better not to do this as it has a tendency to smooth out the general curve.
- I changed the stopping condition from their condition to the distance between the current segment and the end being less than the mean segment length.

As for the rest of the scene, simply have two spheres on top of two cylinders. The spheres are lighter then in the last exercises and you can make them pewter. The cylinders are blue plastic but fade in and out with bright blue every three seconds. The base plane is the same dull green as in the previous exercises. In addition, the spark will reset to a new random spark every quarter of a second.

For those who have read about random walks, this algorithm (using a simple single branch lightning, the “Downward Leader”) is really just a 3-D targeted or controlled random walk.

The one in the images was created with a maximum segment angle of 60 degrees and a mean segment length of 0.01.



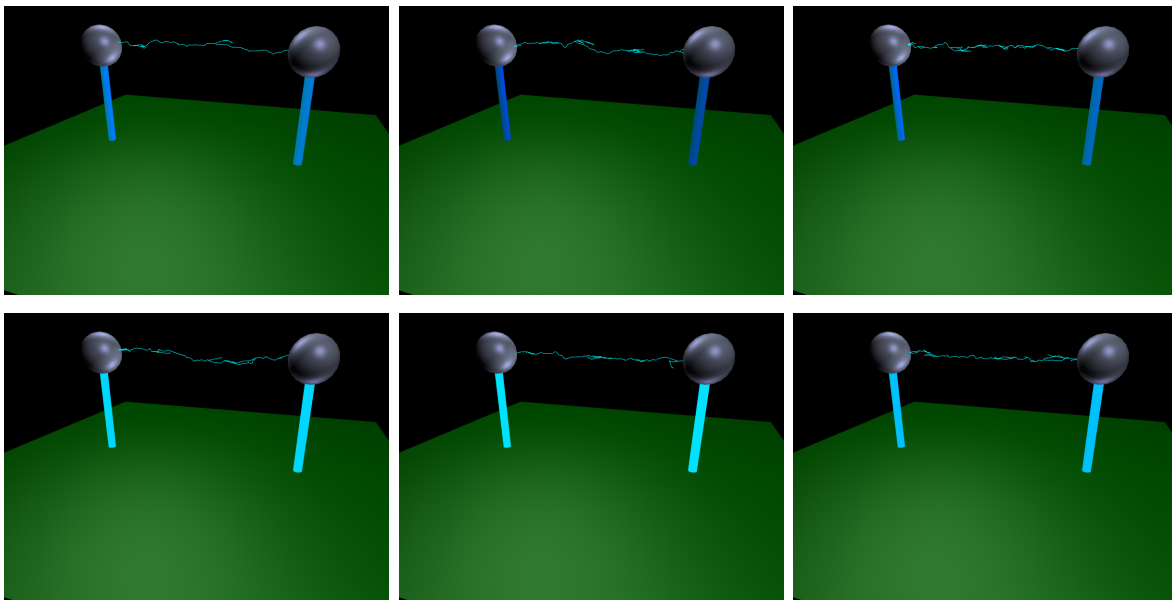
6.2 Exercise #5b

The second stage is to add in branching. This can be done easily by making the random walk recursive. So in the construction of the path you create a random number between 0 and 1 and test to see if that number is less than the probability of creating another branch. If it is then you create new beginning and ending points and recurse into the creation program again. I also included a parameter that tracked the level of recursion and kicked out at 6, that is, I allowed 5 recursive calls at most. Specifically,

- The class constructor will need to add in parameters for the branching probability, the maximum branching angle and the mean branching length.
- In the recursive construction function.
 - It brings in as parameters the starting and ending positions, the segment length, the branching probability, and the recursion level (which I start at 0). The initial values are those given from the class constructor.
 - If the recursion level is above 5 we simply return and add no more branches.
 - If our random test number is less than the branching probability we do the following.
 - * Create a random rotation vector.
 - * Create a random rotation angle between negative the maximum branching angle and positive the maximum branching angle.

- * Rotate the current direction vector around this vector by the angle.
 - * Create a random branch length that is between 0 and twice the mean branching length.
 - * Scale the rotated vector to this length and add the current segment position. This will calculate the end of the branch and the current segment position is the start of the branch.
 - * Now recurse into the function again using the calculated start and end points for the branch. the recursion level should be increased by 1. The segment length parameter is halved on each recursive call as is the branching probability.
- One other change I made was in the initial call to the recursive function, which is not made in the recursive steps. Since this branching depends on the direction of the spark, and the spark could move in either direction, we randomly switch the direction half the time.

The one in the images was created with a maximum segment angle of 60 degrees, a mean segment length of 0.1, a branching probability of 0.1, a maximum branch angle of 20 degrees, and a mean branch length of 0.5.



6.3 Exercise #5c

This addition is an easy timing animation. The program will randomly select an amount of time the spark is on, between 0.25 and 0.5 seconds. Then a time it is off in the same range, then back on, and so on. This will simulate a spark between the two spheres but continuously. The spark should still recreate every fifth of a second. If you do not get the branching program to work you can still do this with the first non-branching program.

Visual Simulation and Rendering of Lightning

Samuel Atkinson

Ian Chamberlain



Figure 1, a sequence of frames from a lightning video export.

ABSTRACT

In this paper, we describe a method for visually simulating and rendering a strike of lightning. Using the random properties of lightning in conjunction with the tendency to strike nearby solid objects, a series of segments representing a lightning strike is constructed. Once the lightning shape has been described, it is rendered with a ray tracing algorithm based on the distance between the ray and each segment. Combining these methods, realistic images and videos of a lightning strike can be rendered.

1. INTRODUCTION

Methods of simulating and rendering lightning require more complexity than most other physical objects because of the nature of lightning formation and its material composition. Physically accurate simulation would require known properties of the air and electric fields in surrounding areas as well as the material properties of the endpoints of the discharge [1]. Visually acceptable simulations can instead use known models of electric discharge to estimate the movement and formation of lightning.

Using observed properties of lightning greatly simplifies the calculations needed to render visually appealing lightning [2]. Instead of simulating the physical properties of electrical discharge, pseudorandom generators can be used to generate segmented meshes that appear accurate.

Another important component in the rendering of visually appealing lightning is the actual light generated by the plasma during a strike. In addition to standard light that is generated by the main channel of the lightning strike, there is also a fairly strong glowing effect that occurs in the area around each segment.

2. RELATED WORK

Previous work on simulating and rendering lightning has fallen under one of two categories: physically-accurate or non-physically-accurate. Because computer graphics often strives to recreate or accurately simulate real-world phenomena, physically-accurate simulations utilize a known physical model called the Dielectric Breakdown Model (DBM). Kim and Lin [3] present such a method. Using the DBM for the electric pattern and the Helmholtz Equation for extended animation, a physically-accurate recreation of lightning can be created. On the contrary, Reed and Wyvil [1] present a method for simulating lightning using a random lightning stroke progression without the DBM.

3. LIGHTNING GEOMETRY

Our proposed method of simulating and rendering lightning uses a non-physically-accurate model to create geometry. Using programmer-set properties and probabilities of a lightning strike, we decide how the computer will construct a random strike of lightning.

Nearly all lightning has a jagged appearance and several branches, so we decided to represent a strike of lightning using a series of straight line segments making up a jagged lightning strike. In code, these segments are represented in the `LightningSegment` object.

A strike of lightning usually consists of one main branch, or Downward Leader [4], which extends from the start point to the target. In many papers the Downward Leader is called the Stepped Leader [3]. Smaller branches extending from the leader are also commonly present. All branches, main and extended, are represented using a series of our lightning segment objects.

This proposed method requires that the user provides a start point in the OBJ file of the scene, formatted as follows:

```
l startX startY startZ
```

The Downward Leader begins at the point (`startX`, `startY`, `startZ`) in world space.

3.1 Lightning Segments

A single lightning segment object contains several properties of its own in order to properly represent a portion of a strike of lightning. First, it stores its start and end points in world space using the `glm::vec3` type. Second, it stores its radius as a floating point number. Third, the lightning segment stores a two-dimensional vector of points representing triangle vertices.

The purpose of the start and end points is fairly obvious. Each segment represents a straight line within the larger jagged lightning strike, so the start and end points represent the line. The radius of the segment is stored for rendering purposes. Lightning branches usually become visibly thinner as they travel longer distances, so the radius is stored in order to represent the thinning characteristic. Triangle vertices are stored solely for the purpose of real-time rendering and debugging. Without using the ray-tracer functionality of our program, the programmer can visualize the general shape of a lightning strike in the scene.

3.2 Properties of a Branch

Determining the geometry of a random lightning strike requires

setting several properties which will be followed during iterative construction of a branch of lightning.

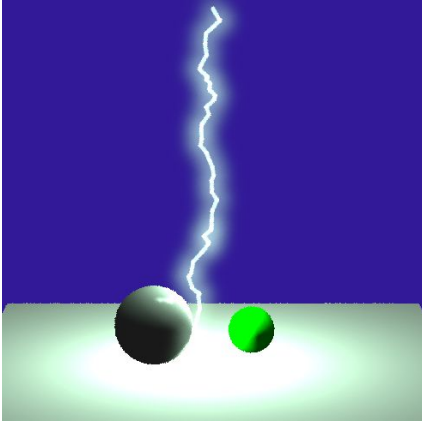


Figure 2, a single branch following set branch properties.

Each branch has independent properties, and the methods for determining the values of such properties are explained in section 3.2.

A single branch of lightning in our implementation has the following properties:

- Starting position
- Direction
- Distance
- Starting radius
- Branch probability
- Mean branch length
- Maximum segment angle
- Mean segment length
- Maximum branch angle
- Rotation normal

The above properties are used to construct a single branch of lightning from a series of straight lightning segments. Values chosen experimentally can be seen in Table 1. Pseudocode for the branch construction algorithm is provided below.

input: the above branch properties
output: a set of LightningSegment objects

```

1.  next <- startingPos;
2.  last <- startingPos;
3.  radius <- startingRadius;
4.  while distance(startingPos, next) < distance do
5.    angle <- random less than maxSegmentAngle;
6.    length <- random with meanSegmentLength;
7.    next <- direction vector rotated by angle;
8.    next <- next * length;
9.    next <- next + last;
10. Add LightningSegment with radius, next, last;
11. HandleBranching();
12. last <- next;
13. radius <- radius - radiusDelta;
14. end

```

A few assumptions are made in the algorithm. In line 7, the 3-D rotation occurs along the rotation normal in the listed properties. In line 5, the angle can be positive or negative; it must be less than the absolute value of the maximum segment angle. In line 6, the length is always positive. It is in the range [0,

meanSegmentLength*2] so that the mean of the length is the appropriate mean segment length. The HandleBranching function takes in the branch probability, maximum branch angle, mean branch length, and the next variable.

The above algorithm ensures that a branch is randomly constructed following the properties set by the programmer. Each iteration of the while-loop creates a single segment in the branch. A single jagged branch of lightning represented by a set of segments is added to the scene if the branch probability is set to zero, and a complex lightning strike is added to the scene if branch probability is non-zero.

3.3 Branching

Branches of lightning are added to the scene in line 11 of the pseudocode in section 3.3. Inside of the HandleBranching function of the pseudocode is a call back to the branch construction function in which it is contained. Recursive construction of the lightning makes it easier to ensure that the properties of each branch are derived from the parent branch and makes it easier to create several levels of branching.

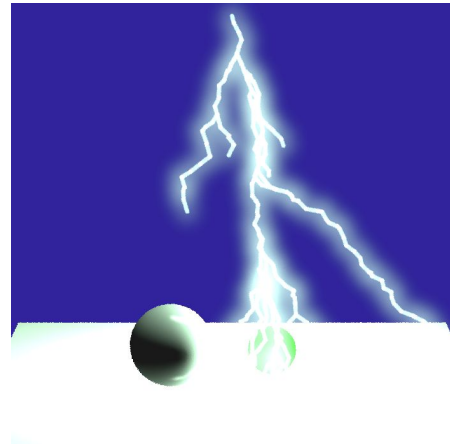


Figure 3, a strike of lightning with several branches.

In order to ensure the construction will not run infinitely, two checks are made before a branch is constructed. The first check is meant to simply obey the branch probability property of the branch. A random number in [0,1] is computed and the branching function only proceeds if the number is less than the branch probability. Second, the branch algorithm checks if the branch probability is greater than a minimum probability constant. This ensures that branches will not continue to be created in low-level branches.

Table 1, branch properties and multipliers.

Property	Starting Value	Branch Multiplier
Radius	0.05	0.5
Branch Probability	0.2	0.8
Mean Branch Length	0.8	0.5
Max. Segment Angle	30°	1.3
Mean Segment Length	0.08	1.0

Max. Branch Angle	50°	1.0
Rotation Normal	vec3(0,0,1)	1.0

After the checks have passed, our algorithm uses a method similar to the provided pseudocode to determine a new angle and distance for the branch. These are computed using the provided mean branch length and maximum branch angle properties. When the new angle and distance is known, a call is made to the main branch construction algorithm to start creating a new branch off of the parent segment. In the new child branch, the branch probability and mean branch length are decreased relative to the parent and the maximum segment angle is increased relative to the parent. Multipliers for constants during the branching process can be seen above in Table 1. These multipliers ensure that the branches are shorter than the parent and that the algorithm will terminate.

3.4 Striking a Target

As mentioned in section 3.1, the Downward Leader of the lightning is the main branch of a lightning strike and determines the overall direction preference of all of the branches. In code, the Downward Leader branch has a decreased maximum segment angle in order to ensure that it will indeed travel the desired distance at the desired direction.

Lightning strikes usually occur between areas of different electric charge, however in our simulation the complexity of setting up such a scenario caused us to rethink how to choose a target for the lightning strike. Instead of always striking the ground or always travelling straight down from the start point, we included an algorithm in our implementation to determine the closest point on a primitive object.

In test scenes, we only used spheres as primitive objects. Our method will check the dimensions of every sphere in the scene, checking the center point in space as well as the radius to determine which contains the closest point. Once a point is known, the direction vector for the Downward Leader is set to a unit vector in the direction of the point. The distance of the leader is set to the distance between the lightning starting point and the closest primitive point.

4. RENDERING

In order to render the lightning, rays are cast out from the camera to render the scene using standard ray tracing methods. There are three main components of the lightning that are added to the final color value for each ray: the light source, main channel color, and the glow effect.

4.1 Lightning as a Light Source

The first significant component for rendering the lightning is to treat it as a light source. Initially, each segment was treated as a point light source, as in [1]. For each ray, then, a shadow ray is cast toward the midpoint of each lightning segment. This method results in fairly convincing shadows, due to the typically high number of segments. In order to improve appearance and allow for soft shadows, our model was extended to use Monte Carlo sampling along the length of each segment. In this case, several shadow rays are cast towards uniformly distributed points along the length of each segment, and the resulting light contribution is averaged for the pixel being rendered.

4.2 Rendering the Main Channel

Each ray's color has a contribution determined by its distance from each segment in the lightning model. This distance calculation is the primary bottleneck in rendering the lightning, since for n segments and r rays cast there must $O(nr)$ distances computed. Section 4.4 describes one optimization, as described in [1], for reducing the number of calculations required in each step.

The lightning contributions for a specific ray can be described as the following [1]:

$$I_{total} = \sum_i m_i e^{-\left(\frac{d_i}{w_i}\right)^{n_i}}$$

With variables:

- I – the main channel contribution for the ray
- $m_i \in [0, 1]$ – the maximum contribution for segment i
- d_i – the minimum distance from the ray to segment i
- w_i – half the width of lightning channel for segment i
- $n_i > 1$ – a “sharpness” value that describes the contrast between the background and the segment

In practice, we used $m_i = 1.0$ and $n_i = 6.0$, with w_i determined according to the radius described in section 3.3.

4.3 Rendering the Glow Effect

Most images of lightning show a softer glowing effect surrounding the main channels of a segment. This glowing effect can be described similarly to the main channel, by the following equation [1]:

$$G_{total} = \sum_i g l_i e^{-\left(\frac{d_i}{W}\right)^2}$$

Where all variables are as in section 4.2, except

- G – the glow contribution for the ray
- g – the maximum glow contribution
- $l_i \in [0, 1]$ – a factor based on the brightness of the segment
- W – half the width of the glow effect

In practice, we used $g = 0.08$, $l_i = 1.0$, with $W = w_i * 3.0$. W was also clamped to a minimum value of 0.08.

4.4 Rendering Optimization

As suggested in [1], this rendering equation can be optimized somewhat by limiting the position of branches to lie in a single plane. With this method, the calculation is simplified somewhat – simply finding the intersection of the plane and the cast ray allows us to compute the point-segment distance with fewer calculations. Although this optimization does not reduce the number of iterations required during the computation, it does reduce the number of operations per iteration. It may be possible to optimize the rendering process further with parallelization or hardware ray tracing.

5. IMAGE AND VIDEO OUTPUT

During the debugging process, we found that rendering using the provided OpenGL ray-tracing result geometry was slower than we would have liked. In order to speed up the process, we implemented direct ray-traced image and video output from the current OpenGL window scene and bypassed rendering the

ray-traced pixel geometry in the window.

Image output was done using the simple PPM image format. When the “y” key is pressed on the keyboard, an algorithm will traverse every pixel in the OpenGL window to retrieve a color value from the scene. Instead of then adding pixel geometry to the window, our method stores the color directly in a PPM file. Conversion from linear colors in [0,1] to PPM RGB [0,255] must take place between ray-tracing and storing. In some cases, we encountered a bug where the linear color was greater than 1, causing a value greater than 255 to be stored in the PPM file. This creates an overflow because the PPM expects a non-negative value less than 255. Before fixing the error, image output consistently appeared similar to Figure 4, below.

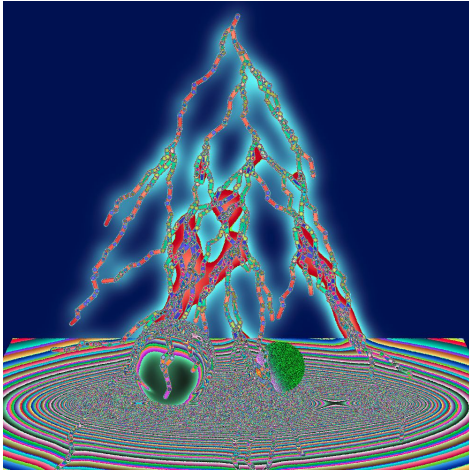


Figure 4, PPM output error.

After recognizing and resolving the error, PPM output works as expected.

Video rendering was accomplished by rendering a series of images and exporting them to PPM. To achieve the effect of an actual lightning strike traveling from start point to target point, the segments generated in the original construction of the mesh are divided into small groups. Starting with zero segments in the scene, an image of the scene is exported using the previously explained PPM method. For each group of segments, the group is added to the scene and a new PPM image is rendered. This continues until all available segments have been added to the scene. In order to convert the series of PPM images into a video file, the following command is executed:

```
ffmpeg -f image2 -r 30 -i ./out/out%d.ppm  
-vcodec mpeg4 -q:v 1 -c:v libx264 -y out.mp4
```

An example of the video output results can be seen in Figure 1. Several frames of the video are skipped in the figure, but it shows the basic progressive construction of the lightning strike shown in a video.

6. CHALLENGES

We encountered several challenges during the development process. While many were small and relatively straightforward to resolve, some prevented development of pieces of functionality.

6.1 Combining Geometry and Lighting

In order to start the project, we decided to take on different aspects of the project. One of us started implementing the

geometry of the lightning and other focused on the lighting and glowing effects of the lightning. Both aspects of the lightning simulation were separated into entirely different code bases and programs.

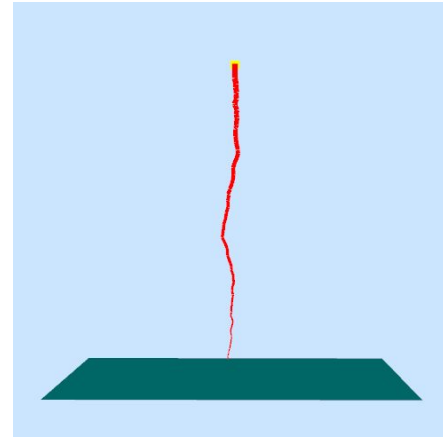


Figure 5, initial geometry program output.

The lightning geometry portion was created as an extension of Homework 2 because it provided a straightforward interface for adding new geometry to the mesh in the scene. It only includes one piece of floor geometry and a point light source at the start of the lightning strike. The start point and direction of the lightning were hard-coded into the program. Output of the initial geometry program is scene above in Figure 5.

The lighting and glow portion of the project was created as an extension of Homework 3. Output of the initial lighting program is shown below in Figure 6. In this example, only the main channel of simple lightning geometry is shown – there is no glow effect applied to the ray tracer yet.

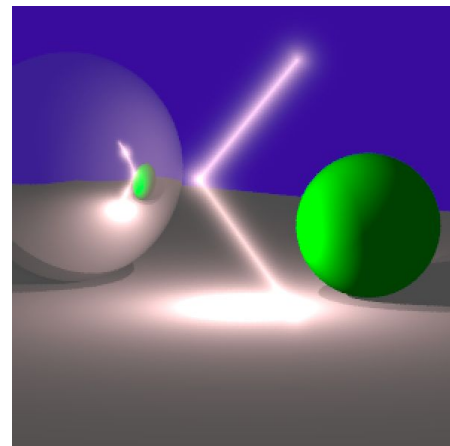


Figure 6, initial lighting program output.

The two portions of the assignment were combined in week 2. Because they were based on different code bases, combination was not trivial. The line segments in the original lighting code appeared in the ray-traced scene but not the OpenGL scene because they were not created as geometry. In the original geometry code, the lightning existed solely as geometry.

Lightning segments in the scene cannot be intersected by the ray-tracer in the same way as other geometry because of the lighting aspect. The width and glow radius of the lightning segments are variable and can be so thin that they do not appear in

the OpenGL scene. For this reason, lightning geometry could not simply be added to the mesh in the scene as originally done.

Instead, lightning geometry is added to the scene only via direct OpenGL calls and is stored in a separate lightning segment structure. When ray-tracing starts, the ray-tracer asks directly for the lightning segments stored in the custom structure instead of polling lightning geometry in scene mesh. The result of combining the two implementations can be seen in Figure 2.

6.2 Creating a Realistic Scene

In order to create a realistic test scene, we needed to use two PPM images as textures in our scene. Using the original code from Homework 3, two images are not allowed to be used as textures. In our first attempt to add two materials to a scene OBJ file, our program crashed at execution with an assertion violation. In the radiosity code, the number of textured materials is limited to one in order to prevent bugs in radiosity lighting. Since we chose not to light the scene using radiosity, simply removing the assertion sufficed.

6.3 Variable Width Segments

In order to give lightning a more realistic appearance, some branch segments must appear to have a smaller width than parent branches. For example, in Figure 3, every branch in the lightning contains segments of the same width. It appears unrealistic because real lightning tends to have visibly thinner branches from the Downward Leader.

We learned, however, that variable width segments do not appear realistic when they are contained within a single branch. In Figure 5, a single branch of lightning appears to start thick and strike the target with thinner segments. This is unrealistic because a Downward Leader always appears to have a uniform width when it strikes a target and electric charge flows through the air.

We achieved variable width segments through modifications in the random geometry generation and the lighting effects. When the lightning geometry is created, each recursive branch is created with a slightly smaller width than its parent. This width is stored in the `LightningSegment` object. When the ray-tracer iterates through each segment of lightning during rendering, it adjust the visible width of the segment as well as the glow radius in relation to the segment object's assigned width.

7. RESULTS

We believe our method of simulating and rendering simple lightning works well and is comparable to previous methods.

7.1 Realism

In order to test the visual accuracy of our method, we created scenes similar to real photographs and placed one of our randomly-generated lightning strikes in the scene.



Figure 7, comparison of real vs. rendered lightning.

Figure 7 shows a comparison between a real photograph of a lightning strike on water and a rendered image using our method which replicates the scene. Notable differences between the images include the purple tint of the lightning in the photograph and the dimmed light emitted by branches far from the Downward Leader.

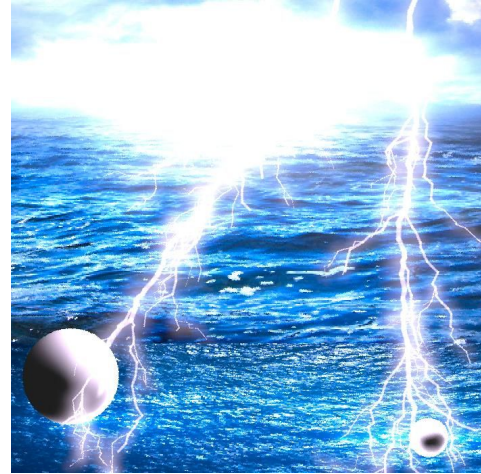


Figure 8, two lightning strikes in an image.

Our method also has the ability to produce scenes with more than one lightning strike. The user must only add another starting position to the OBJ file and a second strike will target the nearest sphere. In Figure 8, two lightning strikes in the scene each target the closest sphere.

7.2 Performance

While sufficiently realistic, our method requires more time to render than comparable scenes from Homework 3. For example, in Figure 8, the scene contains 417 lightning segments. At 700x700 resolution, the scene took 302.87 seconds (just over 5 minutes) to render on a computer with an Intel i7-4600U. Our method is not parallelized, so rendering uses only one core of the CPU's available four.

Rendering an animation is similarly computationally-expensive, but the first few frames of the animation render relatively quickly due to the absence of most branch segments. In tests, an animation consisting of 45 individual images took 48 minutes and 51 seconds to render entirely.

In both image and video rendering, we used the PPM export method described in section 5 to output to storage. Compared to OpenGL pixel geometry originally used in Homework 3, our render times improved. Our results for pixel geometry vs PPM rendering time are shown in Table 2, below.

Table 2, compared image rendering times.

Image Resolution	Pixel Geometry	PPM
100 x 100	3.75s	3.10s
300 x 300	27.73s	20.90s
600 x 600	167.51s	95.06s

7.3 Known Bugs

While our program has never crashed in its final form, there are some bugs that impede the ability to insert our lightning into any scene.

At the start of the geometrical construction of the lightning strike, the closest primitive in the scene is chosen as the strike target. Currently, our implementation only works with spheres or manual directions. If there are no spheres and the programmer does not provide a direction for the lightning to travel, it will always travel towards the origin of worldspace.

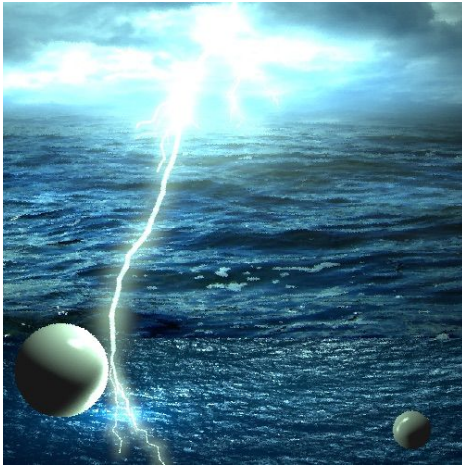


Figure 9, a lightning strike with not enough branches.

Random numbers also cause bugs in our implementation. During some random iterations of mesh construction, nearly no branches will be formed due to the tendency of our random number generator. This can be seen in Figure 9. During other iterations, the lightning strike will consist of too many branches at a high and unrealistic density.

The nature of the rendering algorithm adds lighting contributions from each segment in the image. For this reason, overly bright images can be obtained with fairly small modifications to the scene parameters. Additionally, the segmented nature of lightning typically results in very hard shadows, even when several shadow rays are cast uniformly along each segment.

8. CONCLUSION

We believe our method of simulating and rendering lightning is visually realistic. Lightning and glowing effects of real lightning show in our implementation and the geometry is randomly created, as in real lightning. Our method also has the functionality to render animations of lightning strikes and supports multiple strikes in a single image. Although it is somewhat slow to render, we believe there is room for improvements in optimization in the future.

8.1 Division of Work

Most of the work on this project was split into two parts: geometry and lighting effects.

Sam worked on the lightning geometry, image and video output,

and scene creation. He spent about 15 hours on the project, not including this paper.

Ian implemented the rendering and raytracing aspects of the project. This involved extending the existing ray tracer to account for lighting along lightning segments, adding channel light and glow, and optimizing the ray tracer calculations. His contribution totalled about 12 hours.

8.2 Future Work

While our method does produce visually realistic results, it does so slowly. The more lightning segments are in the scene, the longer the rendering process takes. Future work on this project would focus on optimizing the ray-tracing process to improve render times.

To improve on realism, a few modifications could be made. First, the random number generator could be improved for consistency in branching and angles. Second, the branching algorithm could be extended to allow for more smaller branches on the end of large branches. This was difficult to implement in our experience because the program would either run forever or take several minutes to create the lightning geometry, consisting of sometimes hundreds of thousands of lightning segments. Third, the lightning geometry algorithm could be improved to allow for higher probabilities of branching at different parts of the Downward Leader. Last, the target selection algorithm could be improved to work with more primitive shapes than just spheres.

The rendering optimization described in [1] helps improve rendering time somewhat, but does not significantly reduce the number of iterations required in each ray tracing step. Unfortunately, since the lightning components are not mesh objects, many standard optimizations for ray tracing cannot be used to improve performance of this algorithm. However, it may be possible to greatly improve performance by parallelizing the contribution of each segment.

9. REFERENCES

- [1] Todd Reed and Brian Wyvill. 1994. Visual simulation of lightning. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (SIGGRAPH '94). ACM, New York, NY, USA, 359-364. DOI=<http://dx.doi.org/10.1145/192161.192256>
- [2] Theodore Kim and Ming C. Lin. 2007. Fast Animation of Lightning Using an Adaptive Mesh. IEEE Transactions on Visualization and Computer Graphics 13, 2 (March 2007), 390-402. DOI=<http://dx.doi.org/10.1109/TVCG.2007.38>
- [3] Theodore Kim and Ming C. Lin. 2004. Physically Based Animation and Rendering of Lightning. Proc. of Pacific Graphics 2004. DOI=<http://dx.doi.org/http://gamma.cs.unc.edu/LIGHTNING/>
- [4] Kong, X., X. Qie, and Y. Zhao. 2008. Characteristics of downward leader in a positive cloud-to-ground lightning flash observed by high-speed video camera and electric field changes. Geophys. Res. Lett., 35. L05816. doi=<http://dx.doi.org/10.1029/2007GL032764>.