CPSC589 - Modeling Project

Edraelan Ayuban, Scott Saunders, Cory Jensen

April 17, 2018

1 Overview

Our group has implemented an interactive 3D generator of clouds and lightning bolts.

Users can quickly add and grow clouds in a 3D space. The clouds will merge together by themselves and can also be added to randomly. A simple rendering of the clouds is provided for editing.

In regards to lightning, users can manipulate a B-Spline to define the pathway for a lightning bolt. Lightning bolts are generated with a plethora of random variables, which includes subsequent lightning bolt segment lengths, chances of creating another lightning branch based on height, the angle at which the next segment is placed, etc. Rendering the lightning bolts are accomplished by ray tracing implicit line primitives defined by every individual lightning segment. When given user control, this is mostly implemented in 2D.

2 Implementation

2.1 Clouds

2.1.1 Cloud march

The algorithm used to generate the clouds was taken from [1]. We had to modify the algorithm for various reasons such as the rendering being too difficult and the speed of the program being too slow. The algorithm in the paper works as follows

1. Place some metaballs in your scene that roughly outline your cloud shape.
2. Run marching cubes on your base shape.
3. On the triangular mesh acquired from marching cubes, put a metaball on each triangle with a random position and radius. These metaballs should be smaller than the original.

4. Run marching cubes again.

5. Do 3 again except for the new mesh gained from 4. The metaballs you add should be smaller than the previous round.

6. Go to 4, repeat until metaballs get very small.

7. Perform a volumetric rendering through the meatballs, involving voxelization and other methods for attempting to encompass 3-phase light scattering efficiently. [4]

Steps 3 and 5 are intended to create the wispy edges of a cloud, so the metaballs added here should be quite small.

We made some modifications to this algorithm. We increased the size of the balls added in steps 3 and 5 as we couldn't get step 7 working entirely. To decrease the running time of the algorithm, we added a probability step as to whether a metaball goes on the triangle at all. So the implemented algorithm currently follows these steps:

1. Place metaballs in the scene as you like.

2. Run marching cubes.

3. Add more balls to the scene on some of the triangles.

4. Go to step 2 until you are content.

5. Render.

6. Re-run from step 2 if desired (pushing j).

The rendering for this is accomplished by the inner-product of normals on 3 chosen vectors, up/down/diagonal, and applying a color modifier based on the result. This gives a surprisingly simple but effective basic coloring method for this part of the project.

There were three main metaball functions that we considered. One, is the classic

$$\frac{1}{(x-x_0)^2+(y-y_0)^2+(z-z_0)^2}$$

where (x_0, y_0, z_0) is the location of the metaball. The reason we didn't use this function was that it offers poor blending and has far-reaching removal effects, making all present clouds spherical.

The second function we tried was the Wyvill metaball function:

$$\left(\left(\frac{x}{r}\right)^6\left(-\frac{4}{9}\right)+\left(\frac{x}{r}\right)^4\left(\frac{17}{(9)}\right)+\left(\frac{x}{r}\right)^2\left(-\frac{22}{9}\right)\right)+1$$

, with the piece-wise component of $= 0$ if $|x| > radius$. However, this would not show up in our marching cubes algorithm nor our ray tracing algorithm. We attempted to fix this with linear slopes, as well as $((x-r)(x+r))^2$ for the

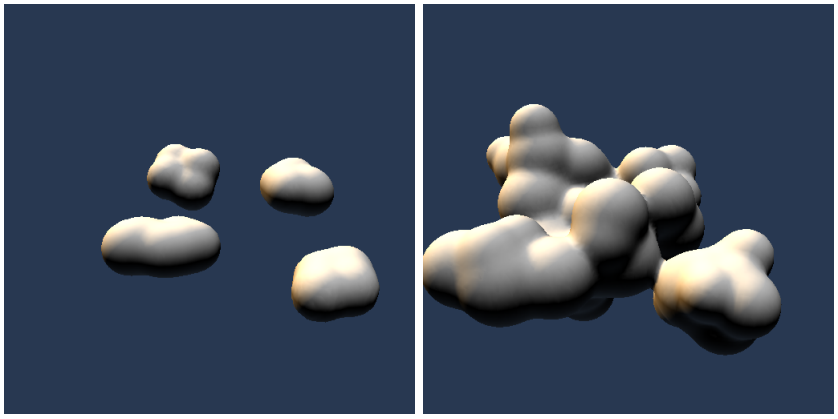


Figure 1: Left: The clouds is just after placing your initial metaballs. Right: The result after marching the cubes 4 times.

out-of-bounds piecewise component, but these still would not render properly in our trials.

The third function we used is based off of Wyvill metaballs - and is what you see in the figures above. It is run without any piecewise logic, which satisfies marching cubes.

$$\frac{r}{-\left(\left(\frac{x}{r}\right)^6\left(-\frac{4}{9}\right)+\left(\frac{x}{r}\right)^4\left(\frac{17}{(9)}\right)+\left(\frac{x}{r}\right)^2\left(-\frac{22}{9}\right)\right)}-1$$

Note: The marching cubes algorithm we are using was an open sourced algorithm that was modified by Dr Sonny Chan, again offered freely. We only modified the existing code for the marching cubes algorithm to get it working in with our program.

2.1.2 Volumetric Rendering

In order to make clouds have varying colors and textures seen, it is required to do some form of volumetric rendering, even if it is only done for blending the edges of the cloud with the background.

This is done, naively, by simple ray-marching. Stepping and detecting if we are inside/outside of the implicit function, at discrete points over some interval along the ray. This method was chosen for it's pure simplicity, yet there was several issues encountered regardless. There was an attempt to accelerate this using various forms of gradient decent, however the floating point rounding issues encountered in GLSL and non-smooth functions made this difficult and prone to error.

For example, if you look at the meta-ball functions, you will see that both wyvil based functions have a positive lip, which does look quite nice for marching

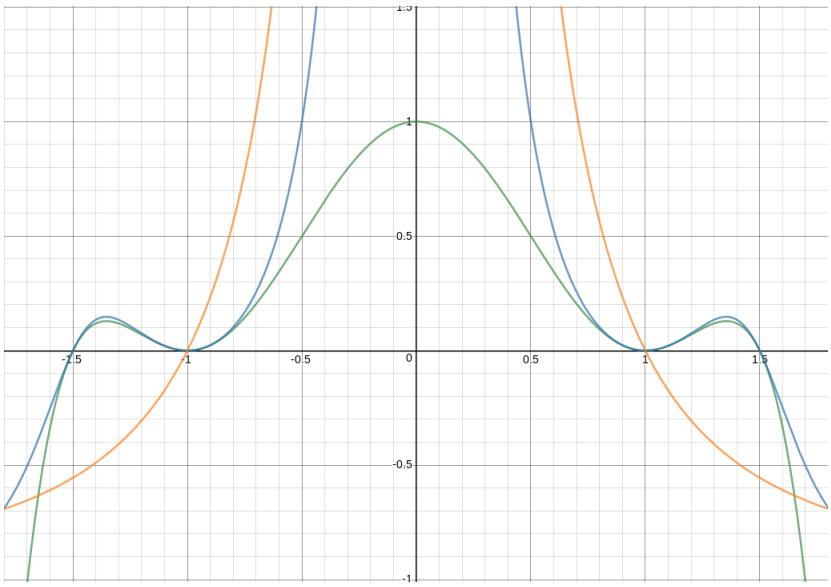


Figure 2: Green: Origonal Wyvill Function, Blue: Modified Wyvill Function, Yellow: Inverse sphere Function

cubes, but gives artifacts in measured linear volume as depicted in Figure 2-left. To counter this, while ray marching we use the inverse-sphere method, giving a stable density/volume, but reducing the fidelity of constructed clouds significantly. This is marked for future work.

Furthermore, the simple effects provided in [3], which we based our rendering techniques only effectively work in forced-perspective, as they effectively texture the object with a simplified colour pattern that varies with depth. This results in images with darkened rings throughout the clouds, as seen in figure 3-right, that, as it is only depth dependent, follow the user as they move about.

The end result was to adapt the [3] rendering just for it's blending function, and merge that with the normal-based coloring of the non-ray traced method. Eventually, the methods depicted in [4] could be adopted to speed-up and improve the quality of the rendered clouds.

2.1.3

Regardless of the rendering method, normals were still required for better lighting effects, and attempting to find the analytical derivative of a summation of varying terms would be difficult. To simplify this we use the numerical method of central-difference, or finite difference. This allows us to quickly get a derivative and/or gradient as required throughout the space, without having to adapt it for each surface function or term in the summation.

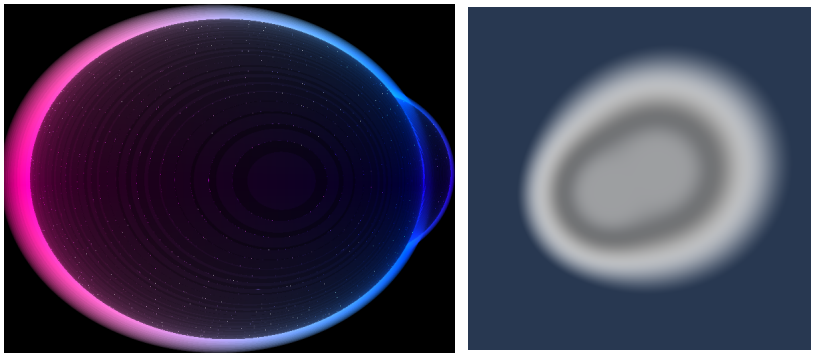


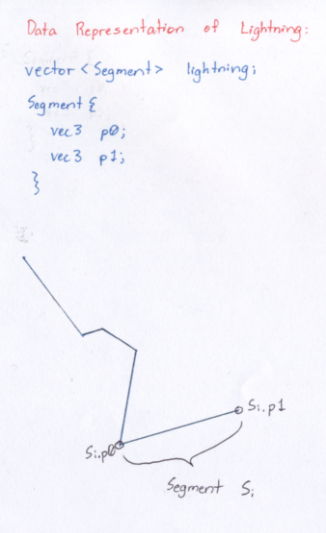
Figure 3: Left: Rendering depth of our Wvil-metaball function. Right: Otogenic rendering on a sphere

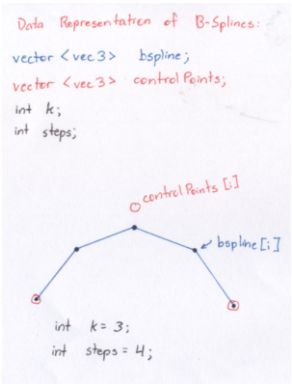
2.2 Lightning

2.2.1 Data Structures

The model of a lightning bolt is a procedurally generated object that boils down into a collection of points in space. The whole lightning bolt is held within a vector that contains a data structure called "Segment". Each segment is a line in 3D space that is defined by p0 (its starting point) and p1 (its ending point).

A B-Spline in our application is an object that consists of two vectors of points. The first vector specifies all of the points that lie on the B-Spline in question. The second vector specifies the control points that are used to generate said b-spline. Two attributes in the data structure control the order of the B-Spline and the resolution of the B-Spline. Both attributes are called "k" and "u_steps" respectively.





2.2.2 Procedural Generation

The first step of generating lightning is specifying the B-Spline path that it should follow. One of the modes allows you to edit a currently existing B-Spline by changing the spline's order, resolution, and manipulating the control points, by adding, removing, or moving control points.

The second step of generation is producing a set of points that define the lightning bolt, following the BSpline. This step is defined as follows:

1. The general direction that we want to head in should be in the direction of the next B-Spline point on the curve. Given the random nature of the generation of lightning, we can be far from the current path. So we calculate the direction by normalizing our current point in space to the next point on the B-Spline.

2. We find a new orientation for our next segment by calculating the Fernet Frame of the direction vector from the last step, and randomly rotating it by the three orthogonal vectors of the current Fernet Frame. (The angle we rotate by is from a normal distribution where the mean is 16 degrees and a variance of 0.1 degrees. This is specified in the paper we have followed [2], who grabs their information from trusted sources).

3. We randomly assign a length to this new direction vector and add it to the current point we are on. This gives us a new point on our lightning bolt. We return to the first step to then calculate our next segment.

The end condition for our loop is how far along the B-Spline path the lightning bolt has traveled. Because the lightning segments can meander around the B-Spline path, we simply cannot compare the two lengths of the curves. The

lightning curve will certainly have the same length of the B-Spline before even traversing the entire path. Therefore, we update the accumulated distance we traveled by projecting our last segment of lightning onto the current segment of the B-Spline. This gives us a better approximation of the distance traveled along the spline path.

Note: Our algorithm above differs from the paper we were implementing in two ways:

1. The path that the paper uses is a linear one, defined by an initial vector given as input. This direction is never changed and subsequent segments are based on the initial vector. We modified this so that the lightning bolt need not follow a strictly linear path, but it can follow one if the input B-Spline is a line.

2. The end condition for their lightning generation is based off of a "ground" condition, in which the lightning bolt will stop when it has reached a certain height. We could not adopt this stopping condition since we could run into an infinite loop condition if we give a pathway that is parallel to the ground. Hence, we use the approximate length metric.

The third and final step of generating lightning bolts is rendering it onto the screen. We implement a ray tracer that uses the formulas given by the paper we were implementing [2]:

- Colour and size of the lightning bolt

$$\sum_i I_{i_\lambda}$$

$$I_{i_\lambda} = m_{i_\lambda} exp - \left(\frac{d_i}{w_{i_\lambda}}\right)^{n_i}$$

  - λ is the red, green, or blue wavelength.
  - I_λ is the light contribution from a segment for wavelength λ.
  - m_λ is the maximum value of a specific wavelength.
  - d_i is the shortest distance between the ray we are tracing and the current segment i.
  - w_{i_λ} is half the width of the lightning channel. w_{i_λ} must be positive and if d_i is much larger than w_{i_λ}, the function is essentially zero.
  - n_i must be greater than 1 and controls the contrast of the lightning channel with respect to the background. The greater the n, the sharper the lightning looks.

- Colour and size of the lightning bolt glow

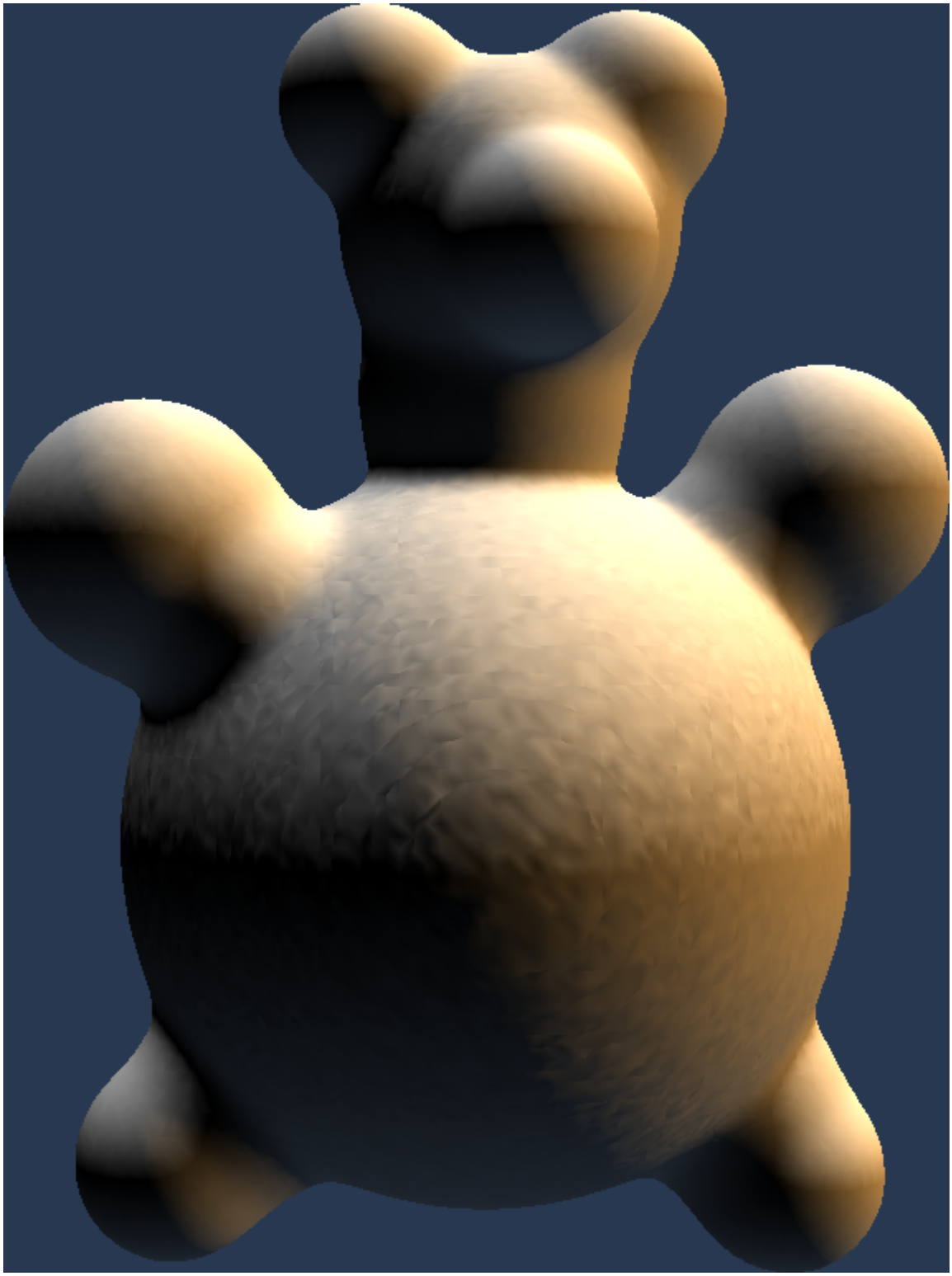
$$\sum_i G_{i_\lambda}$$

$$G_{i_\lambda} = g_\lambda l_i exp - \left(\frac{d_i}{W}\right)^2$$

- G_{i_λ} is the glow light contribution for wavelength red, green or blue.
- l_i is a "life" factor that the paper uses. It describes how far the glow exists from the segment.
- W is the width of the glow if $l = 1$.
- d_i is the shortest distance between the ray we are tracing for and the current segment i

Because of this double sum for each ray trace, and each sum takes into account all the segments in our lightning bolt, the ray trace algorithm is extremely slow for very complex B-Spline path ways.

3 Results



Our end product does not match up with the initial goal that we set forth. Initially, we had wanted realistic looking clouds but have ended up with clouds that look cartoonish, and the volumetric rendering disappointing. The lightening on its own is decent, but without the effects of lightning lighting up clouds, ends up disjoint.

Although there is a method that does merge the lightning with the clouds, there are almost no effects between the two concepts, there is little gained by showing it.

4 Future work

- Placement of lighting with respect to clouds.
- Multiple cloud editing - Saving/loading clouds.
- Multiple lightning editing - Saving/loading lightning/B-splines.
- Proper effects of lighting clouds from the lightning.

4.1 Resolving Known Issues

- When the user changes the meta-ball threshold in Cloud mode, there isn't a visual update of this until a meatball is added/removed.
- There is a lack of z control in the bspline editor mode for lightning, leading to very flat lightning.
- The camera offset when switching between cloud/raytrace modes is clearly disjoint.
- Metaballs don't use the actual wyvill polynomial, but rather a modification that we found to work better within our program, but creates artifacts in density calculation (going to infinity at the center, poor edges)

References

[1] Nishita, Tomoyuki and Dobashi, Yoshinori and Nakamae, Eihachiro, *Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light.* Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996

[2] Reed, Wyvill, *Visual Simulation of Lightning.* SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques. 1994

[3] Gustav Taxén, *Cloud Modeling for Computer Graphics* 1999

[4] Tomoyuki Nishita, Eihachiro Nakamae, *A Method for Displaying Metaballs by using Bézier Clipping* . 1994