Depth Based Toon Point Cloud Rendering

Sean Brown
10062604

Introduction

Goal

My goal for this project is a point-based rendering and uses a pen and ink style illustration method. Exploration of other methods such as depth-based texture modulation and point texturing are other elements of point-based rendering that I wished to explore. Clouds and points are seen in many graphical applications such as games, movies, and other arts. To gain familiarity with rendering points and shading them will be useful in the future for this reason. I wanted to follow the pen and ink style exactly as the paper described but didn't need too. I achieved a similar appearance just by cutting the texture off at various radii. The goal of the project shifted often as I completed each task and then moved on to the next. When I finished the outlined appearance I moved on to depth topics. Currently the end goal is to have a depth dependent cloud rendering system with texture modulation and real time rendering.

The greatest beneficiaries of this project will be those beginning in point-based rendering. I made many choices throughout the project and each affected the outcome and my goal. Using OpenGL's point texture coordinates and not using a hierarchical structure for storing the points all affected my results and I want others to know about these results. Having a finished product means I can reflect and offer the things I learned to others. This applies to myself most of all as learning about point-based rendering was my goal. The topics I learned about in this project have greatly helped me and will be useful in my career.

Previous Work

The first paper I used was *Point-Based Rendering Techniques* by Miguel Sainz and Renato Pajarola [2]. This paper outlines many useful ideas that are found in point-based applications and discusses the best primitives of rendering points, the best data structures to store points, and a comparison of the methods used. The primitives proposed were OpenGL points, Nvidia sprites, and triangles. OpenGL's point primitive was the fastest and simplest of the three and was what I chose for the project. Sprites are provided by Nvidia and are orientable points rather than OpenGL's non-orientable point primitive. The final primitive had the worst performance and are triangles. This refers to the creation of a disk from multiple triangles forming a circle. The octree, Qsplat, and sequential point trees were the data structures compared and octrees were found to be the fastest data structure when rendering points. The paper never expanded on a description of these data structures but only referenced their sources. With some research I discovered that octrees are a data structure that divides space into eight parts. Each node is a divided part and depending on the level of detail required can further divide the child nodes into eight more nodes. This was discovered to be the fastest and most efficient structure for points. Next is the Qsplat which is a program developed by Szymon Rusinkiewicz for large model rendering. It specializes in models consisting of more than 10 million vertices. The Qsplat was the second best method being slower and less efficient than the octree. The final structure tested was the sequential point tree(SPT) this is even less efficient and slower than the previous two structures. The SPT is specialized for points and organizes them into a tree according to the properties they hold. It is very complex but creates a great option for point rendering. The slowest method of storing points was to not

have an organized data structure at all. For the number of points I was dealing with, this didn't affect the speed at all so having no special data structure was acceptable.

The second paper I referenced was *Computer-Generated Pen-and-Ink Illustration of Trees* by Oliver Deussen and Thomas Strothotte[1]. This was a rendering method devised by the University of Magdeburg in Germany as a style for graphical applications. It consists of two parts, the trunk and the leaves. The trunk is treated as a cylinder and drawn using various strokes to shade and shape it. Further branches are more cylinders and are treated the same. The leaves are more complicated and they are the centrepiece of the paper. The system supports any leaf shape desired and can even use multiple shapes for various orientations of leaves. This is for leaves viewed from different angles. The depth buffer is used to calculate the difference between a pixel and it's neighbours and changes colour based upon a threshold. An outline is formed from the depth measurements and stored in a bitmap and finally drawn to the screen. The depth buffer provides a very fast method of calculation which allows for real time rendering. The results of this paper are very impressive and represent trees quite accurately without being photorealistic.

The pen and ink style approach is a successful project that replicated a pen-drawn tree very accurately. It is able to replicate realistic leaves and round point leaves depending on the primitive used and this offers a lot of flexibility. Using the depth buffer offers a boost in performance that allows for real time rendering and also supports large trees. A more realistic approach can be done that isn't real time but can be used for artistic purposes. In the case of the first point rendering paper by Sainz and Pajarola, it's approach is rather successful and offered guidance to my project. Where it fails is in the lack of breadth to the techniques. There are more than four methods of storing points I am sure and maybe they may not be as good as Qsplat and SPTs but they should be mentioned given it is a paper about those methods.

Approach

From these papers I have made choices on my approach and will outline these choices below. In the first paper I learned of various methods of rendering points. The first choice I made was what to represent the points as. I chose the simplest representation which was OpenGL points. They were easy to implement and OpenGL had a built in method of mapping the texture to various positions around the point. Next I had to choose a data structure to store the points. C++'s vector class was a good choice because my data set contained over 800 points which doesn't require a special data structure to render. Finally for the second paper I didn't follow the approach at all. I only used the results as a goal to reach. After I had acceptable success at the pen-and-ink style rendering I moved on to a depth-based problem. This approach isn't based upon a reference and is simple. I solved this problem by dividing the view volume into parts. Depending on the location of a point in relation to a certain depth a certain texture was mapped to it. This results in texture modulation based upon depth.

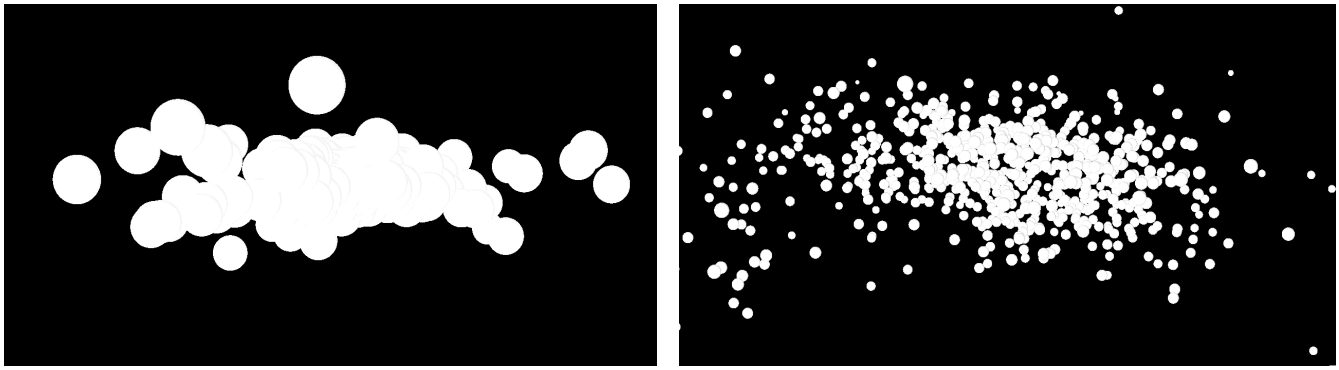
The approach I took supports the time span given to complete the project. I was given four months to plan and implement the entire project and I worked alone. My research uncovered a number of various effects I could choose to explore and as I completed tasks I moved on to new ones. Given the opportunistic nature of the project it is apparent that the finished product is still to be determined. Based on time and the workload, I was very confident I could complete a worthwhile project and learn what I could from the results.

The project will work well because it is completable under the circumstances of a class project. The project's finished state isn't concrete thus the success of the project can still change. Success can be guaranteed if I keep my requirements small rather than large. This project will be successful however because I will ensure that I will complete the previous requirements before adding more. It will also be successful because it will leave a lot of room for expansion which will increase it's chances of success.

Methodology

Rendering Points

I had a few choices before deciding on a primitive to represent a point. These included a single vertex as provided by OpenGL and have static normals, a sprite which resembles an OpenGL point but is provided by Nvidia and have editable normals, and the final choice was a disk made from multiple triangles. Out of these three methods provided by Sainz and Pajarola, I chose OpenGL points because of their speed and efficiency. This choice was also made out of not having access to sprites and the triangle disks being difficult to draw and also naturally slower. OpenGL's points are a built-in flag when calling glBegin and draws each vertex alone without any lines connecting them. Special functions exist that can edit these points but I used the glsl variable gl_PointSize to edit their size. From glsl I was able to add a distance-based point size variation. Using an inverse quadratic relationship I achieved a very natural change in size given the depth of the point. The purpose of adding size variation was entirely to make the cloud more natural looking. Distances were large so I used a smaller percentage of the distance. This percentage and the point size are editable as well.



Left Picture) These example points are large and have a slight outline. This is controlled by the user. Some size difference can be seen. Right Picture) These points are much smaller and the size differences are much more apparent.

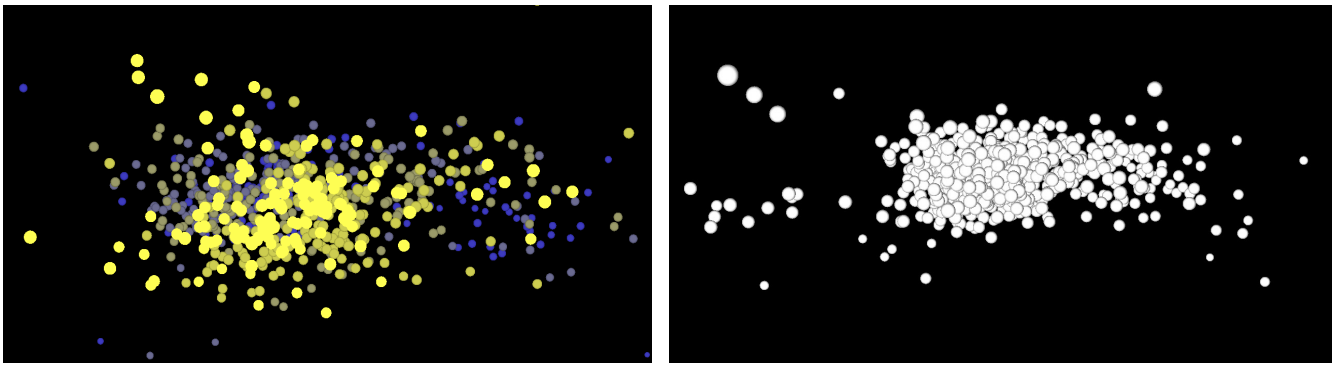
Storing points

The next choice I made was how to store the points. Sainz and Pajarola offered a number of options in this regard including octrees, Qsplats, SPTs, and no special data structure. In my case a simple vector array sufficed and it is easily sorted. The main disadvantage of many of these structures are the effort involved in properly implementing them especially when something simpler works. In performance they may help a lot though and I considered this as well. The kd tree is an efficient option for storing points as well but again wasn't necessary. The vector array offered a dynamic size, easy iteration, and easy sorting. This allows for division of the points according to depth which is done by updating the distances of every point and sorting the points according to distance from the camera. This is the entire process I followed to organize the points within the structure and it has produced sufficient results. I should look at the alternatives I rejected more closely so the decision to use an array is more clear. First of all there is the octree and kd tree which both divide the space and each node contains all the points within each space. This should be compared to the array implementation in terms of finding the points at a certain depth. The array can be sorted and divided by any amount then iterated through by each division. This is simple, fast, and works. To accomplish the same in an octree or kd tree, each node must have a distance associated with it. The tree would need to be updated regularly for the cloud can be rotated and moved. Each node would contain some points and finding a specific point would be

faster because of the tree structure. The main drawback is the complicated nature of the data structure and almost equal payoff compared to an array. I don't need fast access to specific nodes, I need to access all the nodes sequentially. An array is the better solution for it's simplicity. Other structures such as the Qsplat or SPT are fast possibilities but I would need to implement them from scratch.

Mapping textures

The next piece of the project was the texture mapping. In OpenGL there is a built in method of texturing a point with a special vector called gl_PointCoord. This was the only way I could find to map a texture to the point primitive. It treats the point as a square surface with the normal pointed to the camera and maps the texture linearly to the surface. This was my only choice as I could have manually found the positions relative to the point in order to map but that would have been pointlessly complicated. I also found a method of erasing pixels from being drawn in the fragment shader and a formula for cutting the texture into a circular shape. This resulted in the outlined appearance that can be seen when the radius of the cutoff is large enough. Overall the process for mapping the texture was as follows. 1) I loaded all the textures I needed based on the mode of division. Each mode is chosen from the menu. 2) I bind a texture before drawing the points at a given depth and bind the next texture when the next depth is to be drawn. 3) My fragment shader samples the texture using the gl_PointCoord variable. Finally I also cut the texture in the fragment shader by discarding the fragments outside a circular range.



Left Picture) An example of colour change based on depth. This is the five colour option. Right Picture) This shows a similar image to the above but with a much darker outline.

Divide view volume

My final goal was to develop a depth-based variation of textures. A gooch-like appearance was the goal and I achieved this by dividing the view volume and assigning each point to a texture depending on depth. I applied a gradient of two, three, and five different colour divisions and I made sure to make it extensible for more divisions and colours if needed. Dividing the space could have been done in a number of ways. A more complex data structure could be used to store and organize the points. For example, a kd tree could be used to divide the space into close to even numbers of points. I could then classify the depth of each node in the tree and map a different texture to all the points in a node. This seems too complicated because my vector object can easily be divided using simple math and the same effect is achieved. Albeit it must be sorted constantly based on distance as the view changes but the effect on performance is negligible. Dividing the space is really only dividing the points within the structure so that is what I did. Given my points stored in the vector, I divide the space into however many depths I want. Next, I determine the size of each section and then iterate through the vector and

draw the points in each section. This is the simplest solution because it uses a simple data structure and only needs to access each element once per frame.

Results

The measurement of success in this project is the fulfillment of the requirements. The requirements being a point-based rendering featuring a toon-like appearance and depth-based size and texture variation. This can be counted as three requirements and the achievement of each is the measurement of success. In the case of point rendering, success means points have been rendered properly and cleanly. To fulfill a toon-like drawing appearance, they must look as if they have been drawn. An outline and solid colouring are the main indicators of this. Success in depth-based features means that effects are seen that are dependent on depth. In terms of size, I have succeeded if the point size changes based on distance from the camera. In the same area, I have succeeded in texture modulation if the texture changes properly as the points get further from the camera. All of these features should work efficiently and effectively with no reductions in speed.

The requirements I chose to fulfill weren't conducting an experiment of any kind. However they were attempting to create a system that touched upon many fundamental point rendering ideas and have an interesting result at the end. I explored the methods available to OpenGL and followed the advice of past experiences as found in various papers. I would classify this project as an exploration of techniques and effects applied to a point cloud. The features I have completed are the point rendering which features round points of various colours chosen from a menu. A square texture is mapped to each point and cut to become circular. Next I have depth based techniques implemented. Depending on the depth of a point the size and colour of the point will be affected. These are the features implemented.

I will now judge my project in terms of the amount of work done. Over the course of four months I have completed three features that each required a month to complete. Rendering points, texturing points, and depth-based techniques are these features and all are functional. My points are what is expected of the OpenGL point primitive and were simple to draw. Size can be changed through either changing the percentage of distance or directly changing the coefficient. Texturing points is working very well and cutting the texture works as well. Editing the cutoff proves that this feature works. The depth-based techniques are a larger part and can be evaluated as follows. The size variation is the first feature. In my results it is apparent that points closer the the near plane are larger than those farther from it. This is sufficient for this feature. The quadratic relationship between size and distance offers a smooth transition as well. This isn't as easily measured but compared to a linear relationship the quadratic looks more natural. For the texture variation, it doesn't work completely but well enough. The result of texture variation shows a change in colour given a change in depth. This means it does work, however when transformed the points' textures do not reflect the new distances.

These results are an indication that the project was large enough and covered enough topics for the time period. Overall this is a successful project that completes the requirements and explores an area of graphics that is new to me. From the choices I made, the results indicate that my methods were viable and can be expanded to other functionalities.

Discussions

My current results show potential for future work although a lot more work needs to be done. An expansion to the pen and ink style may be promising. If I am able to replicate the algorithm I may be able to get the exact result and the application to clouds would be a truly interesting alternative. Such a

result would have a similar purpose to the original tree application. My depth-based work isn't very compelling because it involves very basic techniques. Their appearance may be well done but it still lacks a purpose. Overall I do find my current approach promising but it will require much more work to become more interesting. It needs more potential and an expansion of ideas.

A better approach to this project would have been to adhere strictly to the reference paper instead of finding my own method of achieving the same result. This would ensure a good result. After finding that I could replicate it through texture mapping I moved onto new requirements. It would have been much better to focus on the proper application of pen and ink on the clouds. However moving on to new requirements during the project helped it grow. A better approach to depth-based techniques would be a reference method to emulate. This would serve as a better starting point than simple texture and size variation. An impressive result would be more likely in such a case. I think this is the second best improvement that can be made to my project aside from the pen-and-ink style rendering. Making mistakes is normal so these are the mistakes I would like to learn from.

To follow up my results I would like to implement the pen and ink style from Deussen and Strothottes' paper according to their algorithm and compare their method to mine. This would provide a test and afterward I can find more depth-based point rendering algorithms to implement. I could even implement a variety of data structures and depth-based algorithms and compare the efficiency between them. This would extend the paper by Sainz and Pajarola which could be worthwhile. Their paper didn't cover every data structure so testing the missing ones would be worthwhile. I would also like to support more file types. As of now I only support csv files.

This project taught me the foundations for point rendering and non-photorealistic styles that can apply to point rendering. Before this I had never explored this area of rendering and throughout the project discovered the most recent techniques and basic knowledge required to complete this project. I gained experience in OpenGL and learned about octrees, Qsplats, and SPTs which are the fastest data structures. By exploring previous papers that related to my interests I was able to choose the most relevant and learn from there. I got to practice the process of determining requirements, planning, and implementing a whole project for the first time as well. Knowledge of this process will surely help me in the future.

Conclusion

To conclude this report I want to repeat that this project has always been about point-based rendering and implementing a worthwhile combination of features to complement it. I explored texture mapping points, editing size and textures, and reproducing an outlining technique. These come together to produce complete project but an improvable one too. I approached this project as a student and offer what I learned to other students. If any wish to work in point-based rendering I may point them in the right direction. Specifically I would advise the use of OpenGL's points and to store them in a data structure suitable to their needs. A vector shouldn't be used in most circumstances but in my case it worked. Point rendering can be a simple method to implement but many choices need to be made which makes it difficult sometimes.

References

1. O. Deussen and T. Strothotte. Computer-Generated Pen-and-Ink Illustration of Trees. Frames and Influence People". Computer Graphics forum (Eurographics '94 Proceedings). 455-466. 2000. Web. http://artis.imag.fr/~Cyril.Soler/DEA/NonPhotoRealisticRendering/Papers/p13-deussen.pdf.
2. M. Sainz and R. Pajarola. Point Based Rendering Techniques. *Computers & Graphics*. 28.6 December 2004: 869-879. Web. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.3831&rep=rep1&type=pdf.
3. R. S. Wright Jr., N. Haemel, G. Sellers, and B. Lipchak. *OpenGL SuperBible Fifth Edition Comprehensive Tutorial and Reference.* Boston: Pearson, 2011, Print.

Acknowledgements

I would like to thank Mario Costa Sousa for his help throughout this project. He always guided me in the right direction and was very open to my ideas which I appreciated. I used some of his code for storing the bmp files as well. I used the gltb class for mouse rotation which was written by David Yu and David Blythe but provided by Mario and I thank them for it.