An Investigation of 3D Graphics Methods

Ramesh Srigiriraju
Computer Systems Lab 2006-07

June 14, 2007

Abstract

The purpose of this research project is to find a way to maximize the speed of a 3D graphics program. To change the runtime speeds, I used different methods to store the matrices used for graphics-related operations (such as rotations, translations, etc.). One storage method involved a matrix expression tree where all the points were recalculated with each rotation and were stored as column vectors. My second storage method stored data as row vectors, while the third calculated the points once at the beginning of the program. The final version that was tested hard-coded the formulas and avoided expression trees altogether. Another area of focus of my program was Z-buffering. After running the Z-buffer algorithm on a set of points, I experimented with four different methods of graphing. The first graphing method simply plotted the points that were visible. The second one drew triangles between the pixels, and the third method combined the first two. The fourth method, which used different schemes for different regions of the graph, worked almost as well as the third.

1 Introduction

1.1 Scope of Study

The scope of this program is to allow the user to graph functions of two variables. The program uses homogeneous coordinates and matrices to perform rotations on the graphs while viewing. The function and the bounds of the viewing window are inputted by the user. In order to store the function, the

program creates a binary expression tree and substitute in values of the independent variables to determine the value of the function at various points. I used this program to test the speeds of the various data storage methods.

Another area of my program involved a matrix editor, which I used to test my matrix expression tree class. This editor allows the user to set the size of a matrix and input data into the cells. After editing the matrix, the user can then perform various operations on it, such as matrix multiplication, Gauss-Jordan elimination, matrix inversion, etc.

After creating these two programs, I used them to test the data structures. The first data storage scheme that I used involved a matrix expression tree. This data structure is similar to a binary expression tree, but it stores matrices instead of numbers. The tree would be used to store the matrix expressions needed to rotate my graphs, and it would be evaluated whenever I needed to plot points. The original, unrotated points would be recalculated each time the viewing window updates itself to take into account any changes in screen size. The original data points would also be stored as column vectors.

My second data storage scheme was similar to the first one. However, the points would be stored as row vectors instead of column vectors to take advantage of the way Java stores arrays. My third data storage scheme involved calculating the original data points only once at the beginning of the program. Other than that, this storage scheme was similar to my second one. My final data storage scheme involved hard-coding the rotation formulas instead of using matrices. The original data would only be calculated once at the beginning.

The graphing calculator module was also used to test various graphing methods for the Z-buffer algorithm. This time, instead of drawing the graph like a "wire mesh," the function would be graphed as a continuous surface. In order to fill in tears that appeared in the graph, I used four different graphing schemes.

The first scheme simply plotted the data points that were not being obscured by other parts of the function. The second scheme drew triangles between visible points: after the points (x, y, f(x, y)), (x+1, y, f(x+1, y)), and (x, y+1, f(x, y+1)) were rotated, a triangle would be drawn containing these points as its vertices. If any one of the points was being obscured, the triangle would not be drawn. The third scheme combined the first and second schemes, while the fourth scheme used the first scheme for shallow gradients and the second scheme for steep gradients.

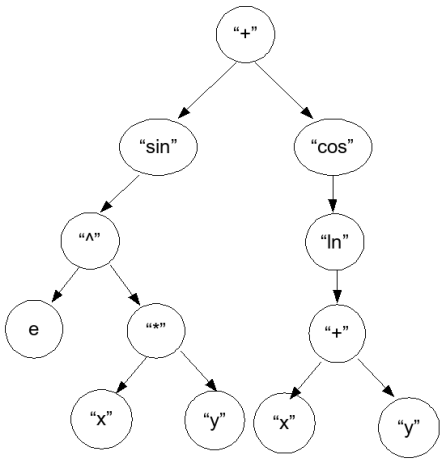
1.2 Purpose/Relevance

The purpose of this research project is to compare different data structures in order to maximize the speed of a 3D graphics program. To do this, I first had to create a binary expression tree class to store expressions. Then, I had to expand this class so that it'd store matrix expressions. Next, I had to create a class that allowed the user to perform matrix operations in order to test my matrix expression tree. Finally, I had to create a class that graphed functions of two variables in order to test my data storage schemes. This research is important to others because it's trying to find a way to optimize the processes involved in 3D graphics. Another purpose is to find ways to reduce the presence of whitespace in a 3D graphics program. The same expression tree classes were used, but the graphing calculator class was modified so that it implemented the Z-buffer algorithm. This area of research is also important to others because it's trying to find a way to optimize the performance of the graphing techniques.

2 Background

Previous projects concerning this area of research include The Investigation of Graphics in the Processing Language by J. Trent and CityBlock Project: Multi-perspective Panoramas of City Blocks by M. Levoy. The 3D graphics projects used rotation matrices, such as the 2D matrix [[cos(a) -sin(a)] [sin(a) cos(a)]], to rotate graphs by an angle a (Levoy, Trent). However, they didn't seem to indicate how these matrices were stored. Other sources specific to Java programming suggested the use of the format xB instead of Ax for linear transformations, where both the column vector and the matrix get transposed. The purpose of this was to take advantage of the way arrays are stored in Java and to reduce errors (Ameraal). Possible state-of-the art programs could be MatLab or other computer algebra systems or even the 3D-graphing feature of the TI-89.

One algorithm that was used in my program was infix traversal. My binary expression trees consisted of a String and two other binary expression trees. The String represented the operation that was being stored, while the subtrees were the two operands:



In my infix evaluation, the two subtrees would be evaluated, and the results would be used as inputs to the specified operation. In order to create the binary expression trees, I split up my Strings in the reverse of my order of operations. They would be split up first based on addition and subtraction, then by multiplication and division, then by exponentiation, etc. That way, my order of operations would be preserved when I evaluated the trees.

Another algorithm that was used in my program was the Z-buffer algorithm. My program started with a set of discrete points, which would be rotated by certain viewing angles. After the rotation is performed, each point would be mapped to a certain pixel on the screen and then drawn. A Z-buffer algorithm uses an array to keep track of each pixel on the screen. Whenever a point gets mapped to a certain pixel, the algorithm checks the array to see whether the point is closer to the viewer than the point that currently occupies that pixel. In my program, the distance from the viewer is determined by the y-coordinate of the point. If the new point is closer to the old point, the old point is replaced in the array and does not get drawn (Ameraal).

3 Development

3.1 Development Plan

My project uses the staged delivery development process, since I have a different plan for each quarter. Every quarter, I have a specific version in mind that has specific functionality. For the first quarter, I planned to just implement a regular calculator module to make sure my infix evaluation algorithms were functioning properly. These recursive algorithms would be used again for my graphing calculator, since the equations would be read in and stored in binary expression trees before graphing. During second quarter, I planned to implement a matrix editing module since the 3D graphics component required the use of matrices. For third quarter, I planned to actually implement my graphing module so that I could test the various data structures. For the fourth quarter, I planned to test the data storage schemes, modify my graphing calculator so that it uses the Z-buffer algorithm, and find ways to eliminate whtiespace.

3.2 Testing Requirements

The implementation of the binary expression trees was pretty straightforward, so my first criteria for determining success involved the actual parsing of Strings. I had to make sure that the listeners associated with the "Enter" button followed the correct order of operations and created binary expression trees based on that order. To test the accuracy of my program, I used the TI-83 evaluation algorithm as a standard. My second criteria was to make sure the matrix editing panel and the matrix operations panel interacted correctly so that matrices could be inputted without losing data.

For the graphing panel, I had to see whether the program could plot points according to a right-handed set of coordinate axes and apply the relevant matrix operations to rotate the graphs. I also had to measure the amount of lag that resulted for each data storage scheme and see which one resulted in the shortest waiting time. After implementing the Z-buffer algorithm, I had to determine the amount of white spaces in the graphs. My final test was to compare the runtime speeds of the third and fourth graphing schemes for the Z-buffer algorithm.

3.3 Research Theory and Design Criteria

To find out the quickest way to store data, I used four different data structures when graphing my functions. The first data storage scheme that I used involved a matrix expression tree. The tree was used to store the matrix expressions needed to rotate my graphs, and it was evaluated whenever I needed to plot points. The original, unrotated points were recalculated each time the viewing window updated itself to take into account any changes in screen size. The original data points were to be stored as column vectors. My second data storage scheme was similar to the first one. However, the points were be stored as row vectors instead of column vectors. My third data storage scheme involved calculating the original data points only once at the beginning of the program and using row vectors. My final data storage scheme involved hard-coding the rotation formulas and calculating the original points only once.

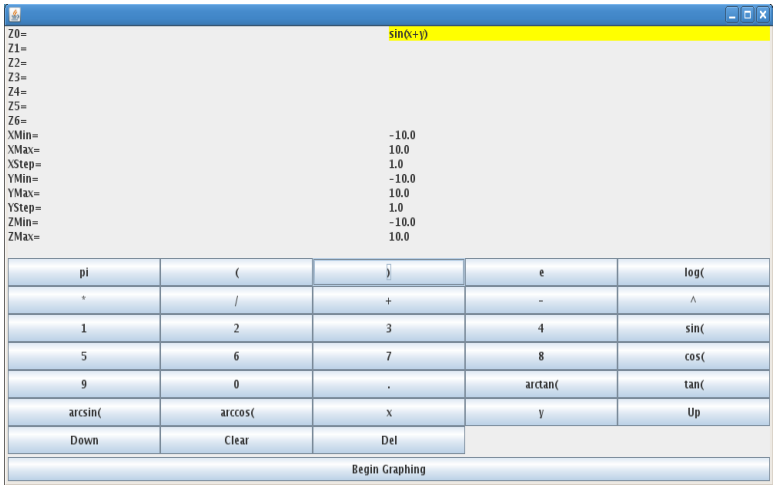
In order to gather my data, I modified one of the listeners for my graphing calculator so that whenever you pressed one of the "Rotate" buttons, it changes the viewing angles and re-rotates the graph 10,000,000 times. Each time, it executes the method System.nanoTime() before and after repainting the graphing window, since it is in the repaint() process that it performs the viewing transformations. Then, it prints out the elapsed time to a text file. After gathering data for each different data storage scheme, I ran another program to analyze this data. This program ignored the first million data points for each scheme, since the programs tended to run slower at the beginning of the trial period and then reached a "steady state" after multiple iterations. The analyzer program calculated the average runtime for the remaining 9,000,000 iterations and printed out the results. I decided to go with 9,000,000 iterations to "dilute" the effects of outliers on the averages. No other programs were running except for the NetBeans IDE v. 4.1. The computer on which I ran my program had 512 MB of RAM and a 1.5 GHz processor. The version of Java used was J2SE 1.5.

However, after I realized that my program was leaving data on the RAM each iteration, and was therefore suffering from memory leaks, I had to modify my program. Each iteration, I had to do a garbage collect to clean up the RAM. Because my program wasn't constantly increasing its RAM allocation, it didn't run as fast, so the number of iterations had to be reduced from 10,000,000 to 10,000.

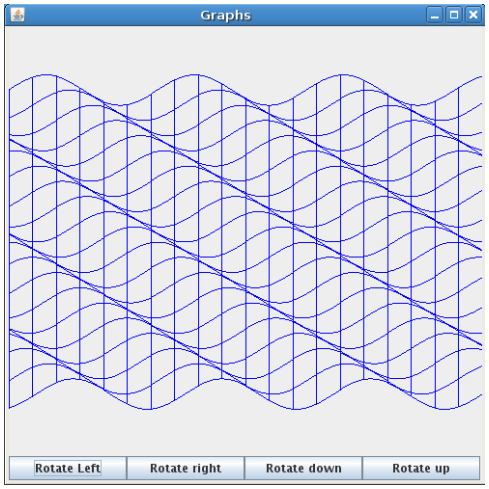
To gather data concerning the Z-buffering schemes, I took a screenshot of each running version of my program and used GIMP to crop out the region from (0, 24) to (501, 525). The screenshots that I took were of the function $z = \dfrac{1}{x^2 + y^2}$ graphed on the domain $x\epsilon[-10, 10]$, $y\epsilon[-10, 10]$, $z\epsilon[-10, 10]$ viewed from the angles $\theta = -.1$, $\phi = \pi/4$. I then used GIMP to convert the screenshots to PPM format. Since the graphs were being drawn in a monochromatic red, the number of white spaces was determined by the number of pixels with nonzero values of blue and green pixels.

3.4 Runtime Process

During the runtime of my third quarter version, the user starts out with this window:



Here, the user can input equations to graph and change the window bounds. After pressing the "Begin Graphing" button, a graph of the function is displayed. The "Rotate" buttons allow the user to rotate the graph in the specified directions.



4 Results, Conclusion, and Discussion

The purpose of this research project is to find a way to maximize the speed of a 3D graphics program. To change the runtime speeds, I used different methods to store the matrices used for graphics-related operations. This research is important to others because it's trying to find a way to optimize the processes involved in 3D graphics. I consider my project to be a success since I obtained enough to compare the performances of each data structure.

So far, I have managed to create a working binary expression trees class that can handle logarithmic functions, exponential functions, trigonometric operations, inverse trigonometric operations, and regular arithmetic operations. The trees for non-arithmetic operations only have one subtree since they only take one argument. I created a class that parsed input Strings and broke them up based on an order of operations that I determined. My matrix editor also uses binary expression trees, except this time, the arguments are matriceds instead of doubles. It can handle addition, subtraction, multiplication, and other operations such as matrix inversion and Gauss-Jordan elimination. I also have a working graphing calculator that can store functions in binary expression trees, apply the rotations that are necessary to view the object, and display the data points on the screen.

The slowest version of my program involved a matrix expression tree where all the points were recalculated with each rotation and were stored as column vectors (Scheme 1). Each iteration, this scheme took 2685 nanosec-

onds to rotate a single graph. Storing the points as row vectors instead of column vectors made a significant difference, since this scheme only took 2513 nanoseconds (Scheme 2). Calculating the points once at the beginning of the program seemed to make no appreciable difference in speed, since the runtime per iteration was 2592 nanoseconds (Scheme 3). The change that seemed to make the biggest difference was to get rid of the matrix trees completely and hard-code the rotation formulas, since this scheme only took 2440 nanoseconds per iteration (Scheme 4). I will not include the data points that were collected in this paper, since there are 40 million of them and they took up a total of 240 megabytes.

However, there are anomalies in the data. Scheme 3 took longer than Scheme 2, even though it performed fewer flops per iteration. A closer analysis of the data I collected revealed that the runtimes did not stay constant. At the beginning, each program took over 10,000 nanoseconds per iteration to run. After a few iterations, the runtime length would spike, and then it would decrease to 9,000 nanoseconds. After running at 9,000 nanoseconds, the runtime length would spike again and then plateau at an even lower value. This process continued until the programs reached a steady-state. By iteration number 20,000, the runtimes would alternate between 2235 ns and 2514 ns. An calculation of the mode data point confirmed this operation: Schemes 1 and 2 had modes of 2514 ns, while Schemes 3 and 4 had modes of 2235 ns.

Table 1: Mean and mode runtimes, without the garbage collect

Scheme	Scheme 1	Scheme 2	Scheme 3	Scheme 4
Mean Runtime (ns)	2685	2513	2592	2440
Mode Runtime (ns)	2514	2514	2235	2235

One possible explanation for the spike-and-plateau pattern involves the use of memory. Every time I ran the repaint() method, my program left data in the computer's memory. As the Java Virtual Machine started to run out of RAM, it would "ask" the system for more, which explains the spikes. This increase in RAM allocation allows the program to run faster, which explains the plateaus. One solution to this problem would be to run the garbage collector (System.gc()) every iteration so that the amount of RAM usage stays constant.

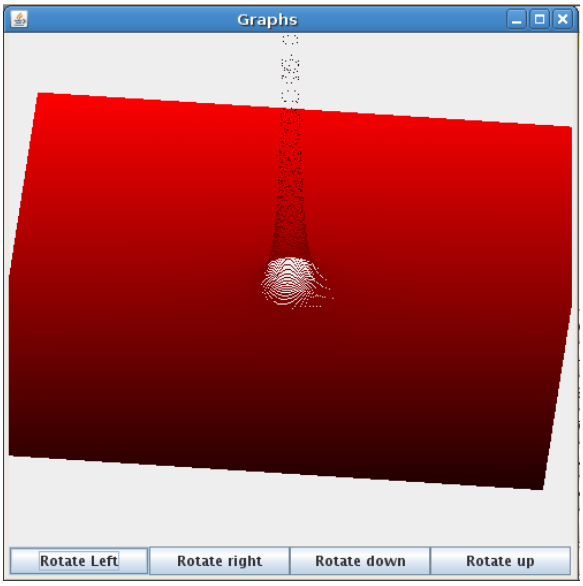
After rewriting my program so that it did a garbage collect every iteration, I redid the data collection. Because memory leaks were no longer a problem, the data didn't show a spike-and-plateau pattern. However, the programs also ran much slower because the RAM allocation wasn't constantly being increased. Because of the decrease in speed, I had to reduce the number of iterations from 10,000,000 to 10,000. The data indicated that optimiation was having the exact opposite effect that it was intended to have: Scheme 1 ran the fastest, while Scheme 4 was the slowest. The average runtimes were 42,558 ns for Scheme 1; 45,343 ns for Scheme 2; 44,679 ns for Scheme 3; and 50,457 ns for Scheme 4. One would think that these results were caused by outliers, which now have more influence on the averages because there are fewer data points. An analysis of the modes disproves this hypothesis. The median runtimes were 33,803 ns for Scheme 1; 36,876 ns for Scheme 2; and 37,435 ns for Schemes 3 and 4.

Table 2: Mean and mode runtimes, with garbage collect

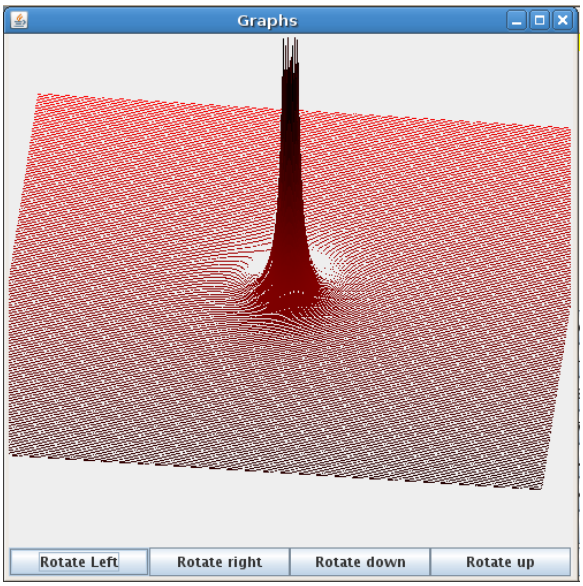
Scheme	Scheme 1	Scheme 2	Scheme 3	Scheme 4
Mean Runtime (ns)	42,558	45,343	44,679	50,457
Mode Runtime (ns)	33,803	36,876	37,435	37,435

Another area of research for this project is to test the effectiveness of different drawing schemes for a Z-buffer algorithm. Scheme 1 just plotted the points that were not being obscured by other points. Scheme 2 drew triangles between these points, and Scheme 3 combined Schemes 1 and 2. Scheme 4 implemented Scheme 1 for regions of the function where the gradient was shallow, and Scheme 2 for steeper gradients.

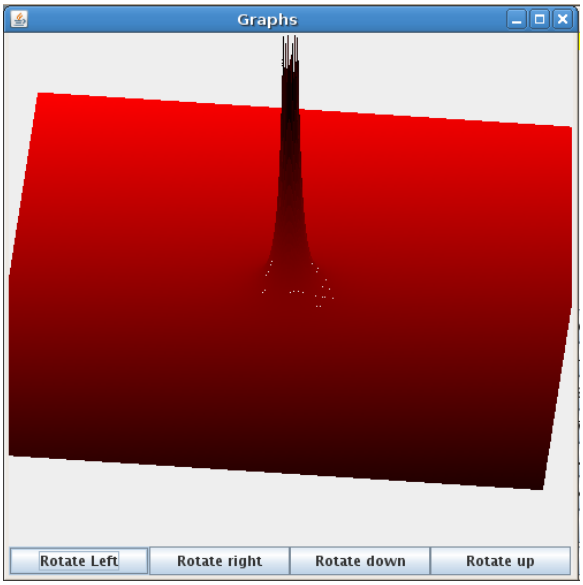
An analysis of the screenshots revealed that for Scheme 1, 72.901 percent of the pixels contained nonzero values for blue and green. Although this can include regions of the screenshot that are "outside of" the function, this also includes white spaces that show up in the graph. Most of the whitespace in Scheme 1 occurred on the asymptote of the function, where the magnitude of the gradient was greater:



For Scheme 2, 88.699 percent of the pixels were whitespace. Most of the white spaces occurred in regions of the function where the gradient was shallower. One possible explanation is that in those regions, it is more likely for a point to be obscured by another point, which makes it less likely that a triangle will be drawn:



For Scheme 3, 65.815 percent of the screen was covered in whitespace. Since it combined Schemes 1 and 2, it worked well for most of the function (both steep and shallow regions). However, a few tears occurred where the magnitude of the gradient was approximately 1:



In order to improve runtime efficiency, I created Scheme 4, which uses Scheme 1 for shallow gradients and Scheme 2 for steep gradients. Because it does not run both schemes over the entire viewing window, it should run faster than Scheme 3, although I did not compare their runtime speeds. However, the measures that I included to save time did not compromise perforamnce, since the amount of whitespace only increased to 66.948 percent:

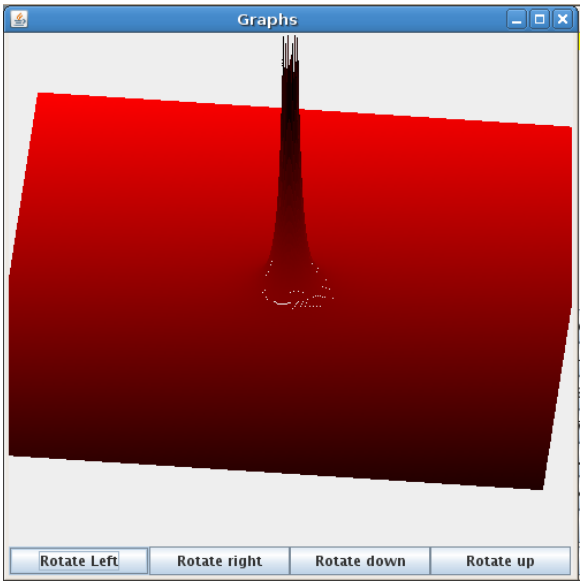


Table 3: Percent whitespace

Scheme	Scheme A	Scheme B	Scheme C	Scheme D
Percent	72.901	88.699	65.815	66.948

My comparisons of the runtime speeds of Schemes C and D confirmed my hypotheses. Scheme C was slower, with an average runtime of 66,742 ns and a mode runtime of 58,108 ns. Although Scheme D had a slightly greater number white spaces in its graphs, it ran faster with an average runtime of 65,159 ns and a mode runtime of 33,803 ns.

Table 4: Runtime speeds for the Z-buffer algorithm

Scheme	Scheme C	Scheme D
Mean Runtime (ns)	66,742	65,159
Mode Runtime (ns)	58,108	33,803

The conclusion that I can draw is that the type of data structure being used has a significant impact on the runtime efficiancy of the program. Although the use of a binary search tree allows the user to write neater code, it results in considerable lag when the program runs. The best way to store a matrix is to hard-code the formulas. Areas for future research would involve

collecting more data to see if the unexpected results of optimization can be repeated. Another conclusion that I can draw is that the magnitude of the gradient determines the success or failure of different graphing schemes. Areas for future research can include testing other schemes and collecting more data.

5 Acknowledgements

I'd like to thank my dad for explaining the "spike-and-plateau" pattern.

References

[1] M. Levoy, "CityBlock Project: Multi-perspective Panoramas of City Blocks," 2006.

[2] J. Trent, "The Investigation of Graphics in the Processing Language," 2006.

[3] L. Ameraal, Computer Graphics for Java Programmers, 1998.