Predictive Insight on the Future of Computer Graphics

Victor Dadfar

Morris Hills High School

March 17th, 2014

Abstract

Computer graphics plays a vital role in modern media. One of the main focuses of computer graphics is the accurate rendering of reality. The purpose of this research is to give insight into the ever-changing field of computer graphics. The procedure includes studying the history of computer graphics, designing and experimenting with algorithms, and developing theories. My final predictions are as follows: the introduction of real-time collision deformation, liquid and hair simulation based off of particle systems with applied physics, an algorithm that uses the molecular makeup of objects to automatically design and build them, and the use of forward ray tracing over backwards ray tracing. These possible developments would bring computer graphics closer to emulating reality.

Literature Review/Rationale

The history of computer graphics dates back to the 1940s, when the premier tool for displaying graphics was the IBM 740 "Cathode Ray Tube Output Recorder" (Belcher), a printer monitor. The 740 crudely output slowly drawn lines in parallel with the material being printed. The refresh rate was non-existent – typically the display wouldn't be cleared until the entire graphic was drawn, which took hours. However, the 740 brought about an important innovation in the field of computer graphics as a primitive, but unrecognized, form of vector graphics, or graphics drawn in a series of mathematical functions. Vector graphics creates lines and other basic shapes by following a set of given functions. The key benefit here is detail, because images drawn using this method could be scaled precisely and indefinitely with no loss of resolution or detail. As years passed, new innovations came about, such as the IBM 780, which had a refresh rate of 20 seconds per frame (or $1/20^{th}$fps by today's standards). However, the 780 sacrificed accuracy for speed.

In 1964, the IBM Vector Graphics Display was released. This announcement acknowledged the importance of vector graphics as a practical approach for computer generated imagery. With a messy but improved refresh rate of 40fps, the display was able to animate objects, and the cursor was soon born. The cursor was the first component in a new wave of graphical user interfaces that would forever become the new standard for user-computer interaction. The type of display used in the Vector Graphics Display was a Cathode Ray Tube. At the time, CRTs were primary type of display output at the time, and though they supported low resolutions, the update rate was far superior to any Liquid Crystal Display prototypes at the time and viable for basic animations. CRTs projected electron guns of different colors onto a fluorescent screen to generate imagery, while LCDs changed the color of individual LEDs,

called pixels, to create graphics. Due to their lower costs and furthered development, CRT displays would go on to become the primary display for users for four decades until the mid-2000s when LCD displays took off.

As the 1970s came around, a new limit was being reached for vector graphics. Although quick to draw, vector graphics were constrained in detail (Belcher). With the introduction of microprocessors and increased memory, a new type of graphics rendering came onto the scene in the form of raster graphics. Raster graphics focused on the color of every individual pixel, rather than the location of functions to draw. Arrays stored the position and colors of pixels, and the increased sizes in storage media made it possible to save these larger files (Fieggen). As a result, efficient solid shading was possible, and computer graphics had enough essential discoveries to grow dramatically. Sprites (collections of images run consecutively to give the impression of animation) and other simple animations made appearances in games. *Super Mario Bros.* (1985) used colorful raster graphics, contrasting greatly with the first Pong of the 1970s and its aged monotonic vector graphics. By the early 1990s, experimentation began on expanding from the second dimension. Until now, flat images and sprites were scaled and skewed to give the perception of 3D. The term ‘psuedo-3D’ was often given to graphics that used projection tricks and illusions to seem like reality (Seidelin). As the power of computing improved, 3D graphics could be rendered another way - using pure mathematics. Ray-tracing (discussed later in detail) and projection matrices were methods of converting abstract object data into tangible projections on a 2D display. A virtual 3D space behind the scenes calculated world variables such as object collision and rendering distance, then projected that data accordingly to the user’s screen through a series of mathematical conversions. Soon enough, 3D engines were built for the masses of developers to achieve the reality of three-dimensional environments with relative ease. These

engines would provide the basis for common aspects between different virtual realities, such as physics, collision detection, and actual rendering.

Physics calculations in engines typically included equations to calculate gravity, rotation, velocity, etc. of every defined object in the scene based on predefined variables. Density, size, shape, and initial velocity were all essential variables needed to initiate an object. Based off these physical attributes, engines also needed to determine collisions between objects, and how the computer would respond to such events. These collision detection algorithms varied from engine to engine, with some sacrificing accuracy for performance. The most basic collision detection algorithms compared positions of every object in the scene with every other object, and set off indicators if any two of these objects shared the same virtual space. Based on qualities of the specified objects in collision, the engine may decide to recalculate the respective velocities of both objects using physics equations, or it may simply ignore the event altogether.

The actual rendering of the objects would be determined from a variety of characteristics of the object, such as position, camera, color, size, and shape. The engine would take in that information and output the respective two-dimensional shapes in a three-dimensional manner. Depending on the type of rendering algorithm used, outputs may be different visually, even if the objects and scene stay consistent. These two-dimensional shapes were referred to as polygons, and over time, polygons reduced in size. Increased computing power allowed for more polygons on the screen at once, to the point where individual polygons can no longer be recognized, especially in cinematic CGI.

The use of 3D Computer Generated Imagery took the place of traditional stop-motion models (Sevo). For example, the 1993 release of “Jurassic Park” was one of the first films to utilize this new technology, and replaced hand-crafted models with computer-rendered clips of

dinosaur herds. Over time the software improved, with advancements such as smooth phong shading implemented for lightning and motion sensors to detect and record realistic movement to use on computer models (Wang). By 1999, movies such as “Star Wars Episode I” and “Titanic” contained CGI that were perceived as unmistakably real, even to the trained eye.

While generated imagery in movies had been consistent lifelike for the last decade, the graphics for video games has not become as realistic. In the late 1990s, the first consumer-level graphics card, GeForce256, was released by nVidia (McClanahan). The advantages of a graphics card included the ability for graphics-intensive applications to offload a majority of visual calculations to the Graphics Processing Unit (GPU) located on the card, and leave the primary processor (CPU) for mathematical calculations. At first, the GeForce256, and other competitor graphics cards utilized “fixed pipelines”, meaning once data was sent into the card, it could not be modified. This resulted in quickly outdated graphics cards, as once a new standard was released, the current fixed hardware could not change to meet those new standards. The early 2000s brought about programmable pipelines, allowing for developers to customize how their graphics card and GPU interpreted and displayed data. Throughout the decade, graphics cards have increased in Video RAM (VRAM) capacity and GPU speed, resulting in more resources for programs to draw larges scenes with more detail than before, and at faster speeds.

Around this time, real-time ray tracing became popular as an alternative and viable method for producing 3D graphics (Rademacher). Introduced earlier, ray tracing was so intensive on the computer’s processor that it could only draw static renderings. Technology improved over time, and the method was revisited. Derived from ray tracing, ray casting was the process of drawing invisible rays to determine when a ray would intersect an object. Ray tracing followed the same principle, but was repeated over and over to find multiple intersections, useful for

calculating reflections and refractions from a light source and objects. With the development of ray tracing, there came two possibilities for the new algorithm: forward and backward ray tracing. Forward ray tracing took rays shooting out from every light source in the scene and calculated individual trajectories, taking into account any reflection or refraction. If a ray intersected with the camera, it would be drawn on screen at the corresponding position where it collided with the camera. Backwards ray tracing was the same logic, but reversed. Every pixel on the screen has a ray associated with it. Those rays were projected from the camera and collided with objects in the scene. At that point, each individual ray's distance and angle from every light source in the scene was calculated and added together. The resulting sums were used to determine total lighting at that individual point. Reflection and refraction was accounted for by reiterating the process over and over again. Both possibilities outputted similar results, but backwards ray tracing was significantly faster performance wise, because there were only as many rays as there were pixels on the screen, versus forwards ray tracing where the amount of rays depend on the amount of light sources, which is quite often much higher.

The field of computer graphics may or may not be possible to anticipate, but it is important to note the current state of the field. Predictions are accurate when the pieces are present that would allow for advancement. Some advancements are spontaneous, and cannot simply be derived from prior discoveries. Others, however, are a continuation in a series of progressions, directly tied to a preceding finding. An appropriate example is the development of shading techniques for two- and three-dimensional objects (Macey). Before this innovative method surfaced in the graphics community, models either had separately-designated shadows meticulously aligned with the actual object they were representing, or the shadows were simply

not there. Shading came as a result of the application of ray casting to lighting, specifically behind objects. The figures below illustrate this method:

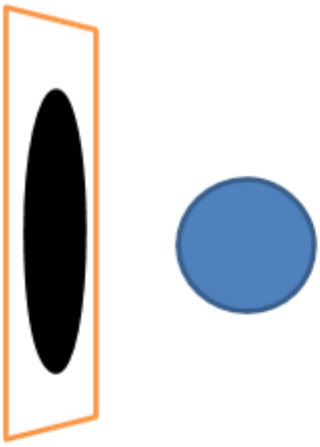

Figure 1A: Shadow drawn independent of object it describes

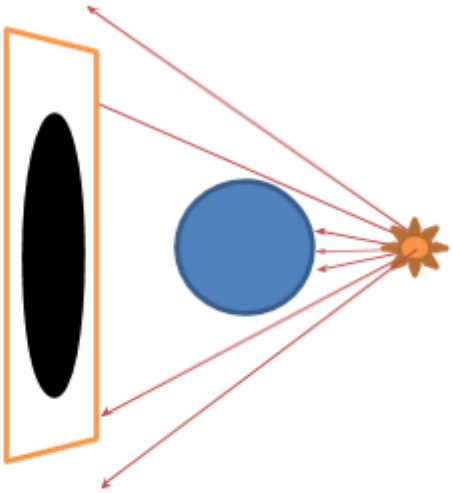

Figure 1B: Shadow drawn as a set of all points not intersected by a ray casted through the light source, rays that intersect an object stop moving

For both figures, the same effect is achieved. However, Figure 1B is not only more accurate (Figure 1A would need to be hand-drawn), but also more efficient, as the shadow moves with the object to dynamically compliment it, whereas in figure 1A the shadow would need to be moved along with the object to give it the same outcome. As a result of this advancement, shading was suddenly a more feasible and realistic feature, able to be incorporated into automatic processes such as rendering and gaming without any human direction. The introduction of

shading through ray casting was very much predictable, as it took an already developed technique of ray casting and combined it with the concept of shading similar to reality. It was another continuation of the technique being applied to different areas. At the time, ray casting was also used for field projections and aiming.

On the other hand, one could take into account the development of the hidden-surface algorithm as a counter-example (Carlson). First demonstrated in the early 1970s, this mathematical equation was the answer to accurate projections of three-dimensional objects on a flat, two-dimensional screen. The explanation was simple – one cannot see surfaces of one object behind another in reality, so the computer should mimic that phenomenon with partial or no representations. Although a fundamental postulate of geometry, it is rarely stated or referred to. Before the publication of this algorithm, objects would always be rendered, regardless of position. Consequently, they were perceived as transparent, not solid. Figure 2 demonstrates the difference in rendering:

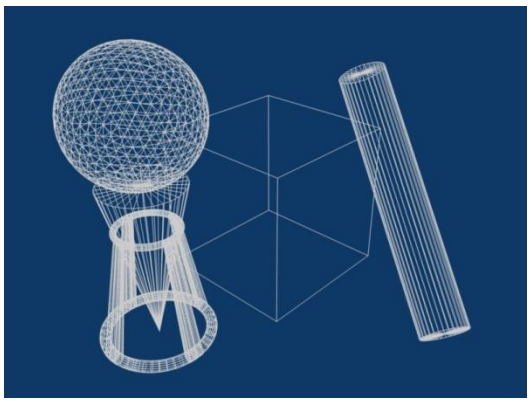

Figure 2A: A simulation of transparent objects, allowing the viewer to see the cube behind the cylinder and sphere when he/she really should not be able to. Objects are also shown to be flat and without substance.

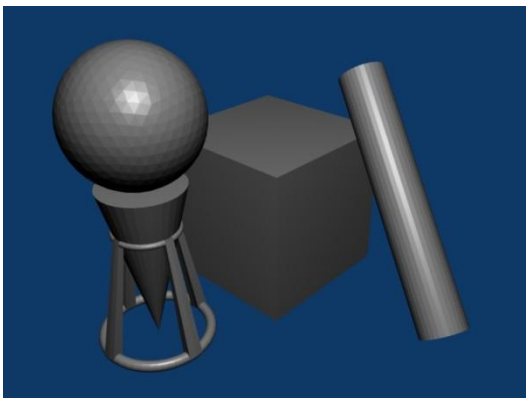

Figure 2B: Using the hidden surface algorithm (and lighting/shading techniques) to hide surfaces that should not be rendered (e.g.) the top of the cone or the edges behind the cube.

A major advancement in 3D rendering, the hidden-surface algorithm was very efficient (drawing one side of an object rather than two), and more importantly, realistic. Computer graphics today would not be where it is if it were not for this algorithm. Beforehand though, it would have been very difficult to predict such a leap in realism because the discovery was so ground-breaking. When 3D graphics were first rendered, people were stunned by this new field in virtual reality, as there was nothing else like it. A researcher working on this paper back in the 1970s would have most-likely failed to make the conclusion that an algorithm would be developed to hide unseen objects. And so, such a prediction was unthinkable for the time, because it was revolutionary. That research in 1999 would have the same problem, and could not conclude that fluid simulation would be so realistic today.

There are two types of predictive knowledge, which will be referred to as obvious predictions and anticipated insight. Obvious predictions are a consequence or application of some existing technology today, and have small implications or benefits. Anticipated insight is off-tangent and mostly grounded in speculation rather than reality, but carries advancements that can dramatically change the field. Both types are equally important, and while obvious predictions are more-likely to occur than anticipated insight, both still have equal merit in this research. This research is important because it leads to educated conjectures that may give insight into the future exploration of computer graphics.

Research Plan

Research Question: Based on the history of Computer graphics, can the future be foreseen?

Hypothesis: As far as Computer graphics have come since the inception of the industry in the 1950s, it still faces the same problem today as it has sixty years ago - the simulation of reality. Although advancements in algorithms and techniques have significantly improved the visual quality of virtual reality, especially in cinematic CGIs, video games and other forms of media still lack accurate realism simulation. This research observes the past and current state of virtual graphics. By utilizing comprehensive knowledge on the history of the field, one can provide reliable insight into the future of Computer graphics.

Materials:

- A computer
- 3D rendering software
- an online course teaching the fundamentals of Computer graphics programming
- Eclipse
- knowledge on selected sources on the history of Computer graphics

Procedure

1. Before any research took place, virtual modeling was experimented with utilizing the rendering software. The knowledge gained from preliminary testing of modern rendering software acted as a hands-on introduction to the field. The different features experimented on were varied shading styles, particle systems, texturing, animation, and physics engines.
2. Following experimental rendering, the sources listed in the works cited were read and dissected to form a comprehensive history on the field.
3. By finding and analyzing patterns in the development of virtual graphics over the last fifty years, obvious predictions on the evolution of computer graphics were made that directly followed these similar and recurring innovations.
4. The series of online lectures pertaining to the computer graphics course was watched. By the end of the course, the knowledge on how to create and customize a graphical shader was taught, along with both the C and GLSL programming languages. These languages were necessary to program the shader.
5. By using both historical knowledge from research and practical knowledge from programming, development began on theoretical algorithms concerning the future of virtual graphics. Reasonable prototypes were created based off these original theories using either the programmable shader, rendering software previously mentioned, or the Java programming language. It is important to note that the algorithms themselves were not created, but rather the general workings, as well as the nature of their effects, were predicted.

Data Collection

The following tables contain data recorder from seven demos testing different techniques to produce similar collision deformation results. All demos include a dynamic object that will not deform, a static object that will deform, and a user-controlled polygon to test the traceability of the static object's deformed surface by following it. These demos were run within a 1000x600 pixel window. The Control group here is a synthesized algorithm currently used in many graphical applications that include collision deformation.

Table 1.A: Qualitative Observations of the Collision Demo Results

Demo #	Draw Type of static object	Shape of static object	Composite Type	Composite Dimensions (Pixels)	# of Composites
Control	Outlined	Planar	Line Segment	1x1	1000
1	Solid	Planar	Rectangle	1x600	1000
2A	Solid	Planar	Square	1x1	600000
2B	Solid	Planar	Square	5x5	24000
3	Solid	Planar	Circle	2xπx50	Varies (Up to 600000)
5	Outlined	Planar	Line Segment	1x1	1000
4	Solid	Circular	Triangle	2x100 (BasexHeight)	360
6	Outlined	Circular	Line Segments	1x1	360

Table 1.A: Qualitative Observations of the Collision Demo Results (Continued)

Demo #	Overlay	Traceable?	Able to produce tunnels?
Control	No	Yes	No
1	No	Yes	No
2A	No	Yes	Yes
2B	No	Yes	Yes
3	Object Path	No	Yes
5	No	Yes	No
4	No	Yes	No
6	No	Yes	No

Column Titles

Composite Type: This is the shape of individual composites that make up the static object.

Composite Dimensions: These are the dimensions of the individual composites.

# of Composites: This is the total number of composites that make up the static object.

Overlay: This is what type of overlay was used in the demo, if at all. Demo 3 uses a circular overlay that records the position of the dynamic object and draws that path over the static object as to give off the illusion the static object is deforming.

Traceable: This is whether or not the deformed static object's surface could be traced by the user.

Able to Produce Tunnels: This is whether or not the demo could account for tunneling during object duration.

Table 1.B: Observations of the Collision Demo Algorithms

Demo #	Description of Algorithm	Major Downsides
Control	Shifts line segments vertically to create hole at impact with predefined specifications	Dent is always of same diameter, overlapping dents
1	Shifts rectangular pieces vertically to create holes at impact conforming to object	Unable to produce tunnels
2A	Deletes square components when hit to create hole at impact conforming to object	Very slow, tracing does not follow tunnels
2B	Deletes square components when hit to create hole at impact conforming to object	Resolution is lower, tracing does not follow tunnels
3	Draws a path of the object to create hole at impact conforming to object	Completely untraceable
5	Shifts line segments vertically to create hole at impact conforming to object and landscape	Unable to produce tunnels
4	Shifts triangular pieces inward to create hole at impact conforming to object	Unable to produce tunnels
6	Shifts line segments inward to create hole at impact conforming to object	Unable to produce tunnels

Table 1.C: Quantitative Observations of the Collision Demo Results

Demo #	CPU Usage (approx.)	Avg. FPS	Avg. Memory Usage (K)	Aggregate Score
Control	13.5%	678.7	43138.0	1165.42
1	22.9%	1324.6	42896.4	1348.43
2A	52.4%	5.9	32589.6	3.45
2B	50.3%	112.9	42967.2	52.24
3	27.8%	65.9	142725.0	16.61
5	19.8%	689.6	41997.6	831.39
4	30.2%	234.3	206156.0	37.63
6	13.8%	679.0	47470.4	1036.50

Column Titles:

CPU: Central Processing Unit

Equations:

Average CPU Usage = $\frac{\sum t}{10}$ where $\sum t$ is a sum of 10 samples of CPU usage taken consecutively

Example: $\sum t = 424, \frac{\sum t}{10} = \frac{424}{10} = 42.4$

Average Frames per Second = $\frac{\sum t}{10}$ where $\sum t$ is a sum of 10 samples of FPS data taken consecutively

Example: $\sum t = 6280, \frac{\sum t}{10} = \frac{6280}{10} = 628$

Average Memory Usage = $\frac{\sum t}{10}$ where $\sum t$ is a sum of 10 samples of memory usage taken consecutively

Example: $\sum t = 456729, \frac{\sum t}{10} = \frac{456729}{10} = 45672.9$

Aggregate Score = $\frac{Avg.FPS * 10000}{Avg.CPU\ Usage * Avg.Memory\ Usage}$

Example: $\frac{628*10000}{42.4*45672.9} = 3.24$

Table 1.D: Quantitative Analysis of the Collision Demo Results

Demo #	Aggregate Score	# of Composites	Adjusted Aggregate Score	Basic Efficiency Rank
Control	1165.42	1000	1165.42	4
1	1348.43	1000	1348.43	2
2A	3.45	600000	2072.97	1
2B	52.24	24000	1253.72	3
3	16.61	Varies (Up to 600000)	N/A	N/A
5	831.39	1000	831.39	5
4	37.63	360	13.55	7
6	1036.50	360	373.14	6

Column Titles:

Basic Efficiency Rank: Ordered by a combination of low CPU and Memory Usage, high FPS, and a high number of composites

Equations:

$$\textit{Adjusted Aggregate Score} = \frac{Aggregate\ Score * \#\ of\ Composites}{1000}$$

Example: $\frac{3.24*70000}{1000} = 227.00$

Table II.A-II.C is an analysis on the different configurations of Demo 2. While Table I.A-I.D dealt with the extreme configurations of Demo 2 (1x1 and 5x5 squares), Table II.A compares all configurations. Since several aspects of these configurations remain unchanged, such as Draw Type of Static Object and Composite Type, they have been omitted from this table for clarity.

Table II.A: Qualitative Observations of the Configuration Results of Collision Demo 2

Demo #	Composite Dimensions	# of Composites	Major Downsides
2A	1x1	600000	Very slow performance
2B	2x2	150000	Slow Performance
2C	3x3	66667	Resolution is low
2D	4x4	37500	Resolution is lower
2E	5x5	24000	Resolution is lowest

Table II.B: Quantitative Observations of the Configuration Results of Collision Demo 2

Demo #	Avg. CPU Usage	Avg. FPS	Memory Usage	Aggregate Score
2A	52.40%	5.9	32589.6	03.45
2B	51.70%	21.5	35448.2	11.73
2C	52.70%	45.8	44574.4	19.50
2D	49.90%	77.8	44688.0	34.89
2E	50.30%	112.9	42967.2	52.24

Equations:

Average CPU Usage $= \frac{\sum t}{10}$ where $\sum t$ is a 10 sample sum of CPU use taken consecutively

Example: $\sum t = 424, \frac{\sum t}{10} = \frac{424}{10} = 42.4$

Average Frames per Second $= \frac{\sum t}{10}$ where $\sum t$ is a 10 sample sum of FPS data taken consecutively

Example: $\sum t = 6280, \frac{\sum t}{10} = \frac{6280}{10} = 628$

Average Memory Usage $= \frac{\sum t}{10}$ where $\sum t$ is a 10 sample sum of memory usage taken consecutively

Example: $\sum t = 456729, \frac{\sum t}{10} = \frac{456729}{10} = 45672.9$

Aggregate Score $= \frac{Avg.FPS * 10000}{Avg.CPU\ Usage * Avg.Memory\ Usage}$

Example: $\frac{628*10000}{42.4*45672.9} = 3.24$

Table II.C Quantitative Analysis of the Configuration Results of Collision Demo 2

Demo #	Aggregate Score	# of Composites	Adjusted Aggregate Score	Basic Efficiency Rank
2A	03.45	600000	2072.97	1
2B	11.73	150000	1759.73	2
2C	19.50	66667	1299.81	4
2D	34.89	37500	1308.34	3
2E	52.24	24000	1253.72	5

Column Titles:

Basic Efficiency Rank: Ordered by a combination of low CPU and Memory Usage, high FPS, and a high number of composites

Equations:

$$\textit{Adjusted Aggregate Score} = \frac{Aggregate\ Score * \#\ of\ Composites}{1000}$$

Example: $\frac{3.24*70000}{1000} = 227.00$

Data Analysis

Collision Demos Comparison with Control

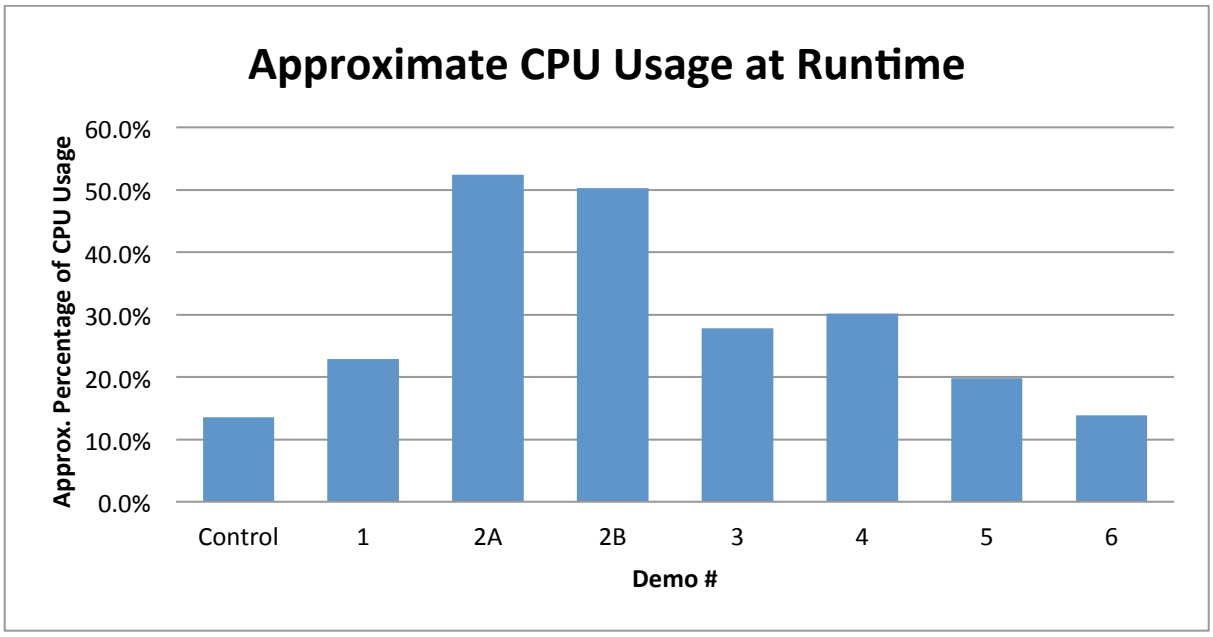

Figure 1: This graph is a comparison of CPU Usage percentages recorded when running the demos. CPU stands for Central Processing Unit, as in the component of the computer that interprets information and outputs calculations, much as a calculator. All demos were run on the same windows machine with no background applications. The CPU Usage was recorded from Task Manager, a Windows application that displays vital computer information like CPU and RAM usage. Higher CPU Usage percentages equates to more work for the computer to process the information and code given by the demo. Programs with higher percentages also tend to run slower on older, less powerful machines, so the demo with the lowest CPU Usage percentage is most favorable to run on all machines. As the graph shows, both configurations of Demo 2 (2A and 2B) require the largest percentage of CPU to run at around 51%, meaning they require more processing power to run. Both the Control and Demo 6 show to use the least amount of CPU

Usage at about 12%, meaning they are favorable over other demos in terms of CPU Usage percentage.

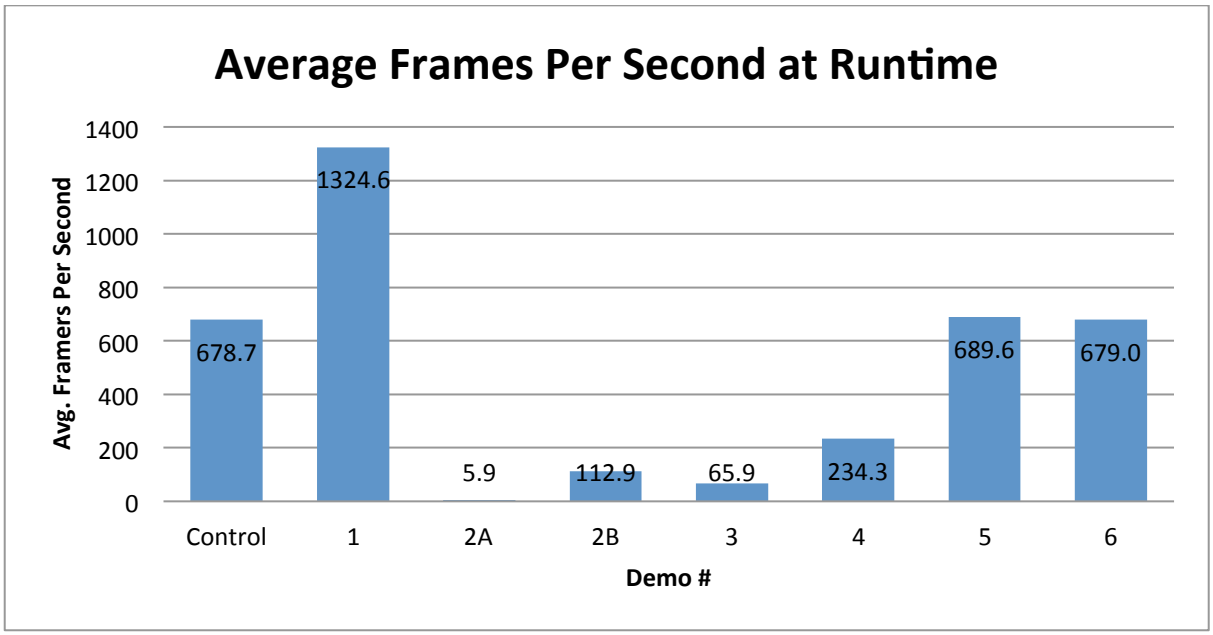

Figure 2: This graph is a comparison of average frames per second recorded at runtime for each demo. The FPS data was recorded from a script added to the demos to display frames per second for the program at a one second interval. Samples of those outputs were recorded and averaged. Demos with FPS values lower than 30 are choppy, while those higher are much smoother. Above 60 FPS, the demos show no difference in performance. For example, Demo 1, at an average of 1324.6 FPS will not perform differently than Demo 5 at an average of 689.6 FPS, regardless of the 600 frame difference. The point of this graph is to illustrate that Demo 2A, at 5.9 FPS, is very choppy and undesirable for a graphics application, while Demo 1 clearly has the best performance, even if it is not noticeable by the user.

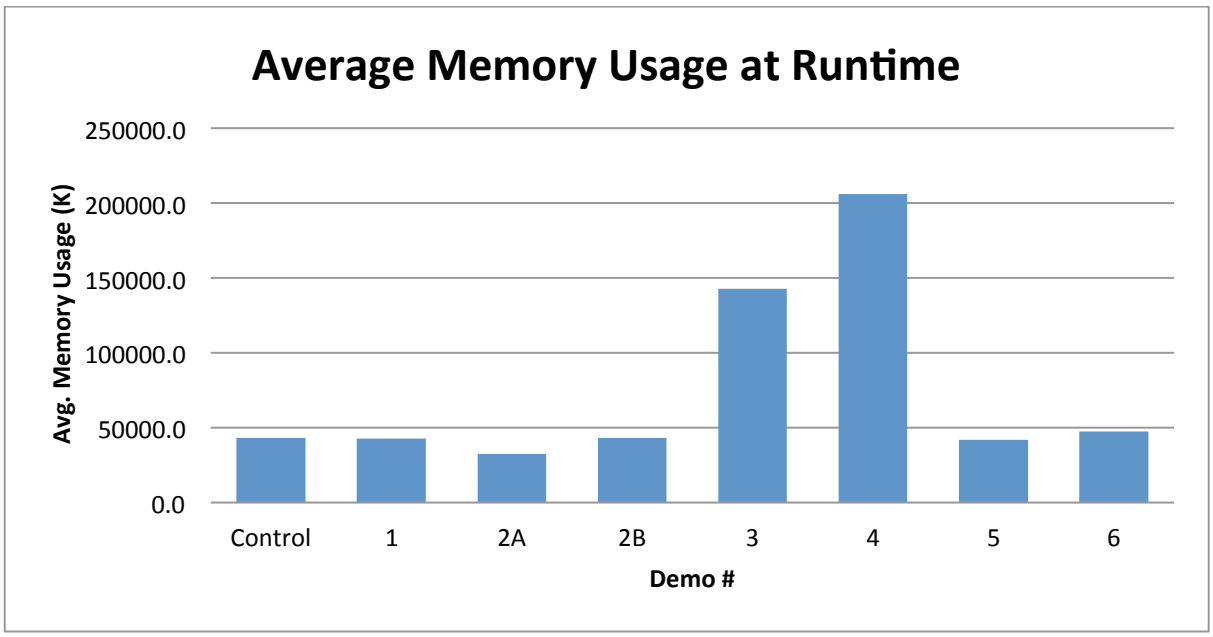

Figure 3: This graph shows the average memory usage of the system RAM at runtime for each demo. RAM stands for Random Access Memory, and is where data is temporarily stored for quick access by the CPU. Lower RAM values are favorable because they are compatible for older machines with a smaller supply of RAM. Data is recorded in KB, or Kilobytes. One kilobyte is 1,000 bytes, and is simply a unit of virtual storage. RAM usage was recorded from Task Manager. This graph shows Demo 4 uses the largest amount of RAM at runtime, while Demo 3 uses about ¾ of that amount. These two demos use much more memory than the other demos, so they are less favorable. On the other hand, Demo 2A and Demo 5 use the least amount of memory, so they are more favorable because they leave a smaller footprint on the system's RAM and require less to run.

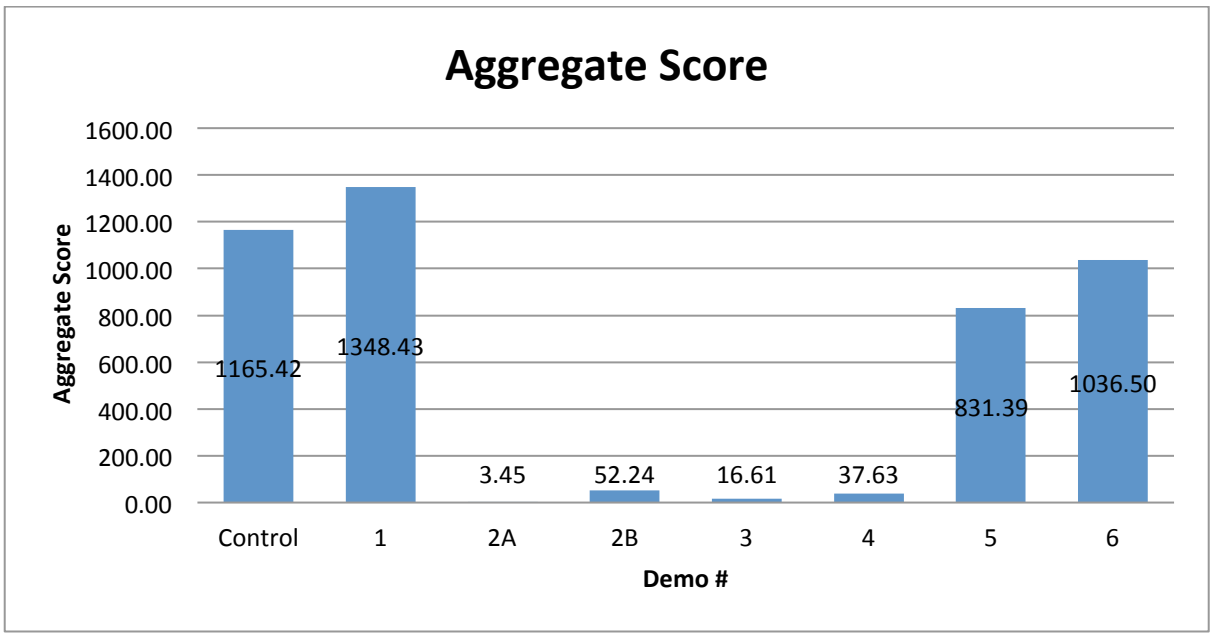

Figure 4: The aggregate score is a compilation of the approximate CPU Usage, average FPS, and average memory usage values for all demos. The equation for the aggregate score is shown in the Data Collection section. The equation is designed to give demos with more favorable data higher scores; in other words, demos with low approximate CPU Usage values, high average FPS values, and low average memory usage values will score high, while demos with high approximate CPU Usage values, low average FPS values, and high average memory usage values will score lower. Demo 1 scores the highest, meaning it has a combination of those favorable values. Demos 2A, 2B, 3, and 4 scored the lowest, meaning they have a combination of the least favorable values.

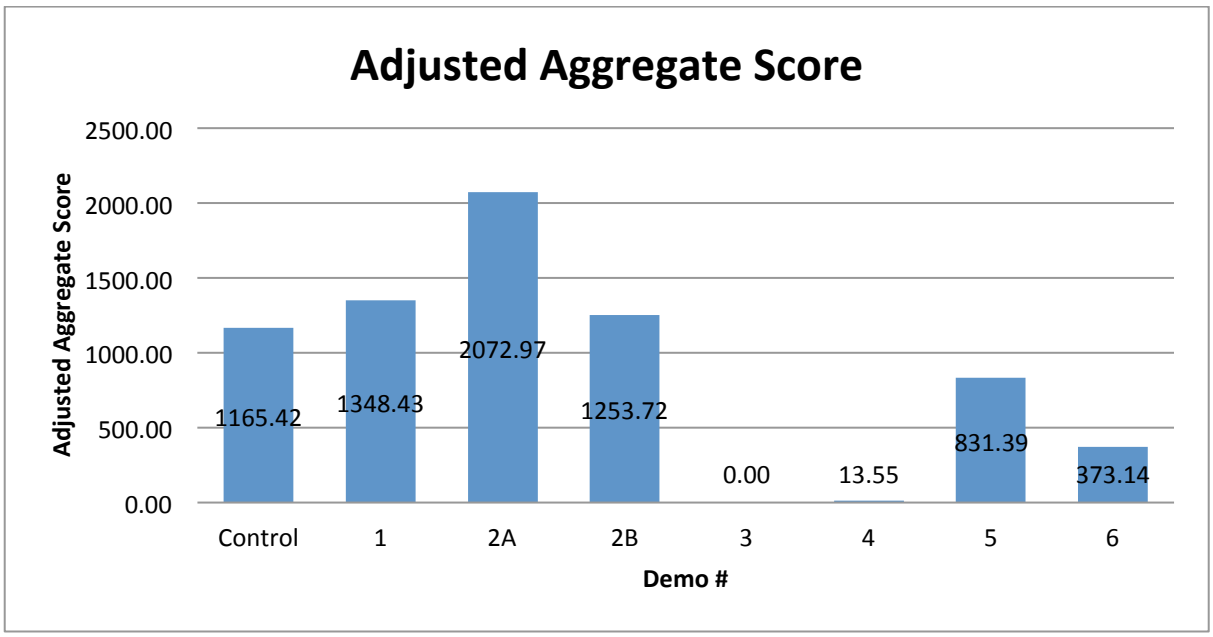

Figure 5: The adjusted aggregate score uses the aggregate score and takes into account the number of polygons each program is drawing, as to give a more accurate comparison. A program that draws more polygons will require a larger percent of the CPU to run, more system memory to store information, and will most-likely result in a lower FPS. In order to give demos drawing more polygons a higher score, the adjusted aggregate score uses that variable of number of polygons. The equation for this score is displayed in the Data Collection section. Demo 2A ends up scoring the highest in this comparison. Even though Demo 2's CPU Usage percentage is highest and the FPS value is low (both unfavorable values), it does have to draw the most amount of polygons at 600000, while other demos typically draw only 1000 polygons. As a result Demo 2A is the most favorable demo compared to the rest. Demo 4 scores the lowest, meaning it is not a favorable demo to use. Demo 3 has no score because the number of polygons it draws varies, so a number could not be used to calculate this score.

Configurations of Collision Demo 2 Comparisons

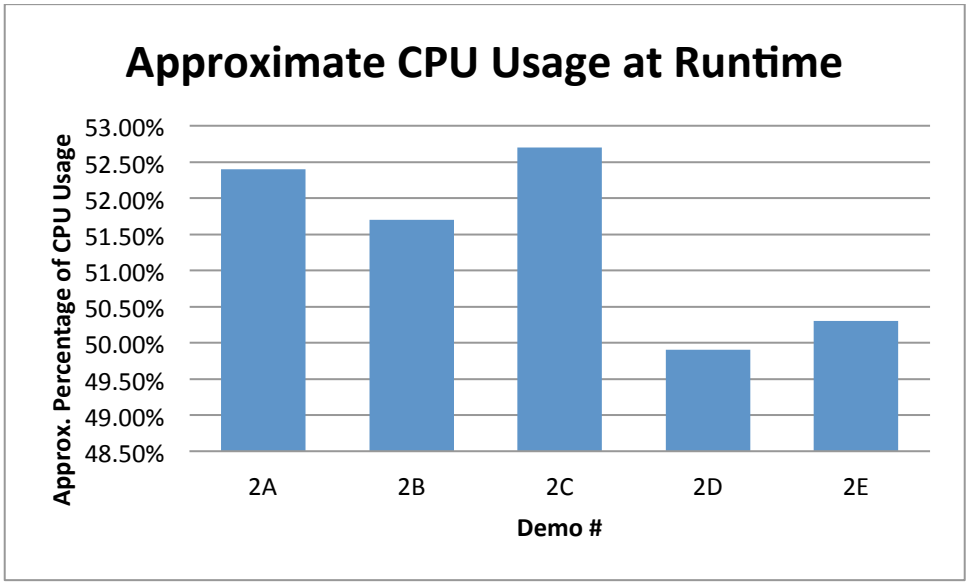

Figure 6: This graph is a comparison of CPU Usage percentages recorded when running the Demos 2 configurations. As the graph shows, configuration C of Demo 2 requires the largest percentage of CPU to run at around 53%, meaning it requires more processing power to run. Both the configurations D and E show to use the least amount of CPU Usage at about 50%, meaning they are favorable over other demos in terms of CPU Usage percentage. The scale is small here, because different configurations of this demo do not significantly affect the CPU Usage of Demo 2.

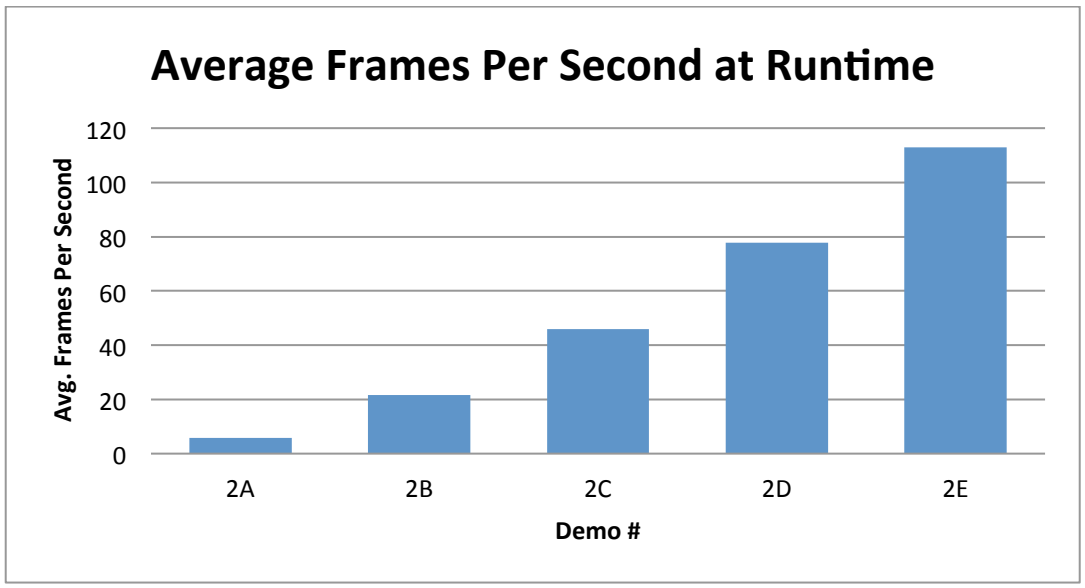

Figure 7: This graph is a comparison of average frames per second recorded at runtime for each configuration of Demo 2. Configuration A of Demo 2, at about 5 FPS, is very choppy and undesirable for a graphics application, while configuration E of Demo 2 clearly has the best performance at about 110 FPS.

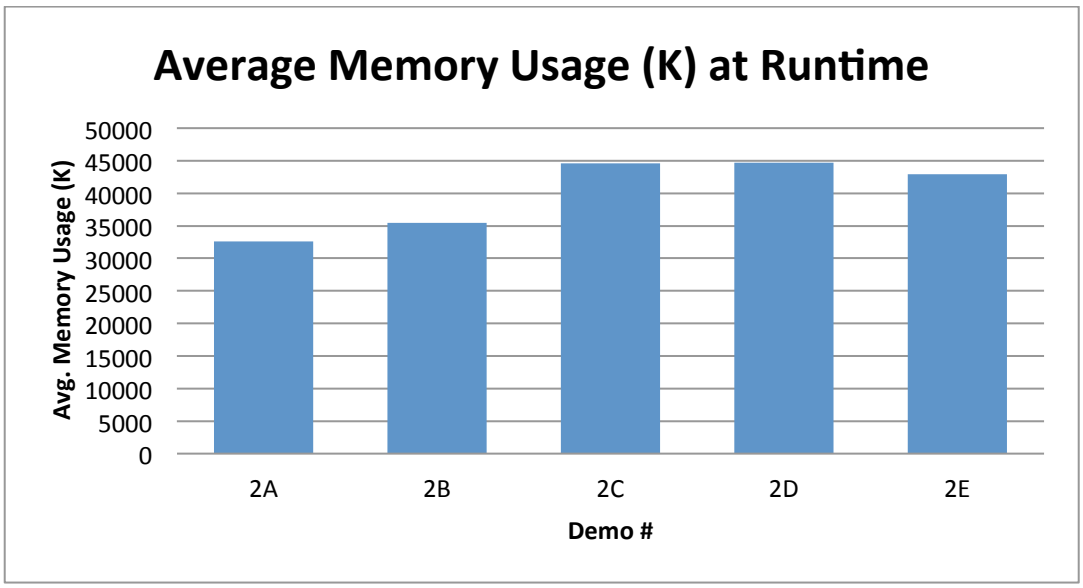

Figure 8: This graph shows the average memory usage of the system RAM at runtime for each configuration of Demo 2. Configurations C and D of Demo 2 use the largest amounts of RAM at runtime. On the other hand, configurations A and B of Demo 2 use the least amount of RAM, so they are more favorable.

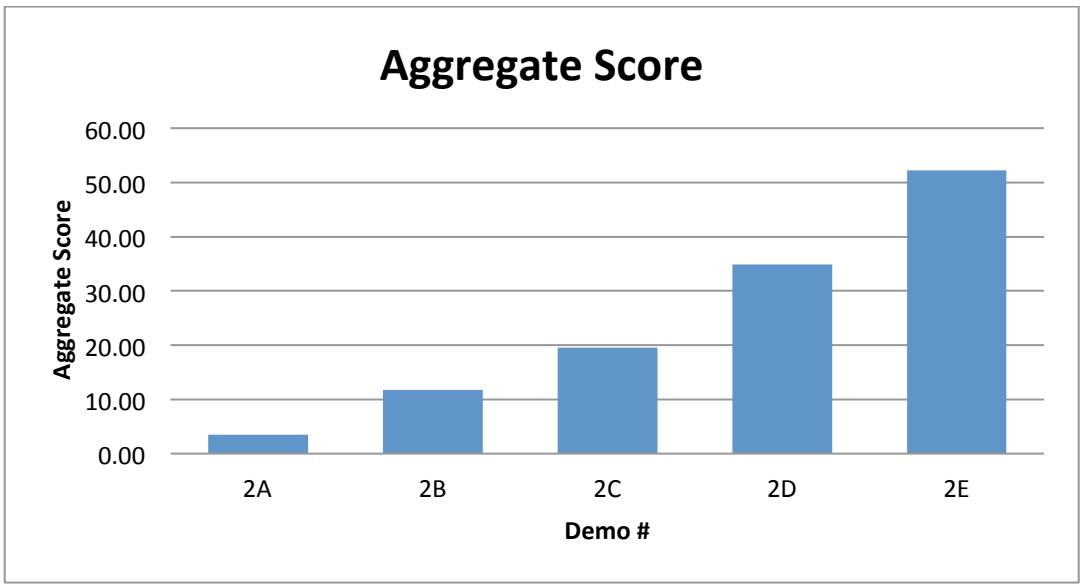

Figure 9: The aggregate score is a compilation of the approximate CPU Usage, average FPS, and average memory usage values for all configurations of Demo 2. Configuration E of Demo 2 scores the highest, meaning it has a combination of those favorable values. Configuration A of Demos 2 scored the lowest, meaning it has a combination of the least favorable values.

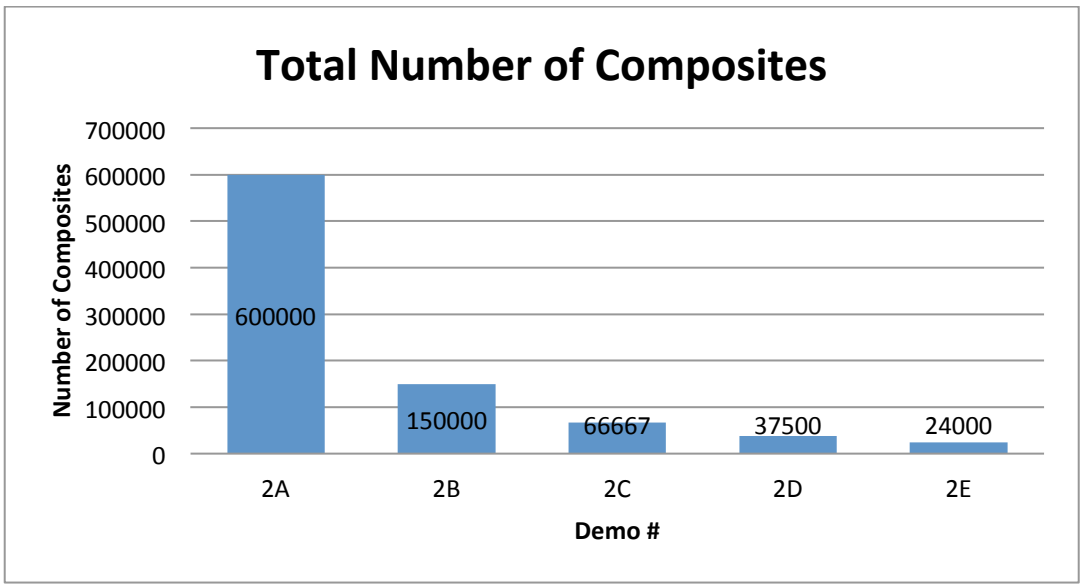

Figure 10: This is a graph of the number of composite polygons each configuration draws at runtime. Configuration A of Demo 2 has the most polygons to draw at 600000, while configuration E of Demo 2 only has to draw 24000 polygons. This data is useful to find the adjusted aggregate score in the next part.

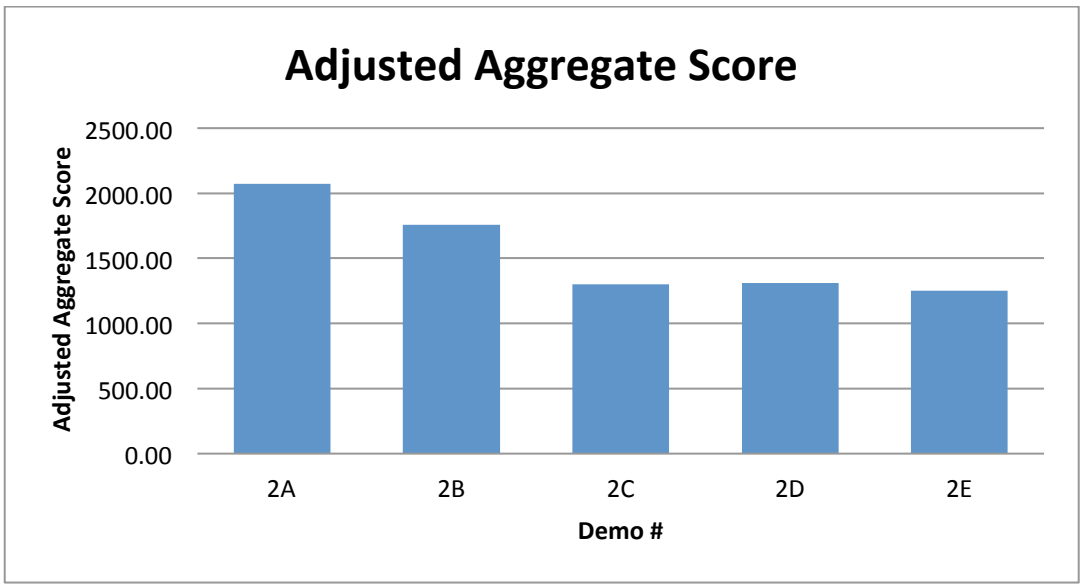

Figure 11: The adjusted aggregate score uses the aggregate score and takes into account the number of polygons each program is drawing, as to give a more accurate comparison. Configuration A of Demo 2 is the most favorable demo compared to the rest. Configurations C, D, and E of Demo 2 score the lowest, meaning they are not favorable demos to use.

Conclusion

The future of Computer graphics is foreseeable. The field has seen frequent advancements over the last few decades, and visuals today are spectacular in detail compared to their primitive counterparts of the past. However, there is still a wide gap between reality and virtual reality, and future innovation will be able to close that gap. My research creating collision deformation, liquid simulation, and ray tracing has shown that the algorithms for a more realistic virtual world are already possible. Since Computer graphics today has two primary applications, cinematic CGI and interactive programs, the findings of this research will either be specific to one of these applications, or to the field as a whole. If Computer graphics is to simulate reality as accurately as possible, it must follow the same laws that govern nature. Insight given from this research aims to combine computational logic with the laws of physics to create the most realistic virtual reality.

Resolution, or the amount of pixels utilized on the screen, has always severely limited the potential of a realistic simulation of reality. The data collected has shown that the cause of the problem is always hardware related, either because the display does not contain enough pixels, or the machine is not powerful enough. If the user is able to distinguish individual pixels, or jagged lines are present where curves should be, the resolution is not high enough to accurately represent reality. Perspective is also an issue for hardware. As an object is farther away from the user, it decreases in size to give the impression the object is actually far away. That object may look smooth close up, but is much rougher from a distance, where the display has to use fewer pixels to draw the same object. At the current pace, an adequate display that can solve this dilemma will never exist, because an infinite amount of pixels would be needed to draw objects at every distance without roughness. However, there is a threshold that, once passed, will

effectively reduce this problem. The human eye has a limit; for a display with a density that exceeds 163 pixels per inch, the retina can no longer tell apart individual pixels. This density is beneficial to smooth jagged lines in fonts and images, but is not high enough to resolve the issue of perspective. Advancements in the production of displays will further reduce pixel width and increase pixel density. 4K TVs have a resolution of 3840 × 2160 pixels, significantly higher than HD displays. Some of these televisions have pixel densities around 150 ppi, which is very close to the retina limit. Over the years this limit will be surpassed, several times over, as hardware capabilities catch up to software capabilities. As the hardware improves, graphics will seem genuinely smoother, and objects can be drawn at a distance without suffering from rough edges. A consistent improvement in display resolution is an obvious prediction that applies to the field in general.

A low resolution may not always be the fault of the display. Quite often, developers purposely run their programs at lower resolution to make up for the lacking processing power. Demos created for this project have had to be run at lower resolutions in some case to make up for otherwise slow machinery. This problem is only in interactive programs; cinematic CGI is rendered from supercomputers over extended periods of time, so processing power is not a limit. Interactive programs need to be able to redraw the display dozens of times a second in response to user input. The higher a resolution is, the more pixels a program needs to account for, and the longer it takes for the machine to calculate those pixel's color values. As hardware ages, developers must lower the resolution of the program in order to keep the program running at an acceptable frame rate (the rate at which the screen is redrawn) and to avoid slowdowns. Instead of having each individual pixel have its own color value, groups of two or three pixels share the same color, reducing calculation load on the processor. Unlike the last issue, this one is mostly

self-solving. Increases in consumer computing power will allow for higher resolution, and therefore higher quality, graphics. The speed of processors has always, and will continue to develop to perform more calculations than the last generation. A consistent improvement in computing power is an obvious prediction that concerns the application of Computer graphics in interactive programs.

Since the inception of three-dimensional graphics, almost all types of Computer graphics use polygons to represent reality. The theory behind this is that every real-life object can be drawn using polygons, generally triangles and quadrilaterals. For interactive programs, these polygonal meshes (groups of connected polygons) are insufficient to represent reality, because they contain finite detail. Cinematic CGI, because it is scripted, does not have to be infinitely detailed to look realistic; rendered scenes need to have just enough polygons to fool the user into thinking there aren't any actual polygons. Interactive programs cannot dictate what the user sees, because they are interactive, so everything must be infinitely detailed. Over time, more and more polygons will be used to describe objects, making them seem even more realistic. But, the level of detail cannot match that of reality using this current method. Mathematical models can be used to perfectly represent basic geometry shapes, such as spheres and planes, as my research has shown they have infinite detail. But these models are not practical to use in drawing an imperfect reality. In our world, real objects are not made of pure polygons, but individual particles at the microscopic level. The types of particles, as well as the bonds between particles, determine all characteristics of the object, from strength to shininess. Computer graphics must approach rendering the same way to truly represent reality. A virtual reality made up completely of incredibly small particles is the best way to achieve realism, and has three immediate benefits over polygonal rendering.

1. Liquid simulation will be very accurate. Currently, liquid in interactive programs are drawn using animated, semi-transparent sheets of polygons. As a result, these sheets are not able to simulate liquid physics realistically enough. Developers use tricks like independent animated splashing to give off the impression the liquid is real, but the result is far from how liquid actually behaves in reality. CGI Artists spend countless time modeling polygons to look like liquid, when they could be spending their time modeling something that needs to be paid attention to, such as a human face. Replacing the polygon model with particle simulation that follows the laws of physics will not only make liquid simulation a true simulation, but also save CGI artists time as they do not have to model liquid by hand. Applying this particle system to liquid simulation, the 2D graphics demo that was a part of this research simulated liquid interaction with foreign objects. The simulation was far more realistic than any current methods to render liquids in interactive programs.
2. Both elastic and inelastic collision deformation will be very accurate. Interactive programs use pre-determined functions to depict dents for collisions, and most objects are static and have no pre-determined functions. The result is a mostly static virtual reality. CGI artists, as with liquid simulation, have the benefit of modeling collisions by hand, to make them seem real. A system based solely off of particles would solve the problem of realistic collisions in both applications. Collision Demo 2 showed that objects made up of tiny particle composites are the most efficient to run, and are the most realistic, because they mirror real-life structures of solids. The computer would take the load of calculating collisions between individual particles, taking into account particle type and bond, to ultimately describe the resulting shape of both

objects after the collision. This approach is most realistic because it follows the laws of physics that make up this reality. And as with liquid simulation, the automatic process saves time for artists who would no longer need to model collisions.

3. Hair and thread simulation will be very accurate. Current renderings use long, thin and flexible columns to visualize hair and thread. Due to the limited interactions between these flexible columns and environmental conditions, such as wind, the threads seem stiff and unnatural. Thread, and hair, composed of microscopic particles, however, will interact with the environment appropriately, because they have the same structure of thread and hair in reality.

Using microscopic particles in Computer graphics is currently not feasible for the most part, but the algorithms are already present. Using the mathematical model for a sphere, individual particles can be drawn with infinite detail, since the particles themselves are drawn from a flexible function, rather than finite polygons. However, the processing power to build a virtual world made up solely of particles is not present in consumer media. It is possible, though, to be incorporated into cinematic CGI, as that hardware has already improved enough for basic realities. As processing power increases in all forms of technology over the years and decades, particle-based renderings will be possible. The introduction of this particle system is insight that applies to the field in general.

Using the particle system to determine object attributes would be favorable, since real-life objects are based off of their molecular makeup. Unfortunately, that would require large amounts of processes that would slow down even the fastest of supercomputers today. The molecular makeup of real-life objects is very small and sophisticated, and technology that will be able to run completely-particle-based systems will not be developed for years to come. In the

meantime, an alternative method to producing similar results may be developed. Rather than calculate the whole particle structures of objects, the program would simply take the parameters of those particles to essentially "build" the object, without actually calculating every individual particle. In other words, given the type of particles and bonds between them, object characteristics can be calculated by the computer. For the developer, time is saved in making both interactive programs and cinematic CGI, as objects are determined by their components, in the same fashion present in reality. The use of an algorithm to calculate object characteristics based off particle traits is insight that applies to the field in general.

Backwards ray tracing has been popular in cinematic CGI for years now, and real-time backwards ray tracing has recently become practicable in interactive programs. A simple backwards ray tracing demo developed in Java as part of this research showed that the hardware performed smoothly when the objects were animated, so even real-time backwards ray tracing is not completely appropriate for this generation of machines. Regardless, for Computer graphics to be as realistic as possible, this method of ray tracing will no longer suffice in the future. Rather, forwards ray tracing will become the standard in virtual reality. This is due to the fact that in reality, light rays are emitted from the light source, not the eye, and bounce off of multiple surfaces before reaching the eye. Forward ray tracing takes the same concept, and applies it to Computer graphics. The expendable power available in the future will allow forwards ray tracing to be feasible for the developer. Because technology in the future will be far more powerful than it is now, forwards ray tracing, which requires more resources, will be possible in cinematic CGI, and eventually interactive programs. The use of forward ray tracing to calculate light is insight that applies to the field in general.

These expectations cannot be guaranteed. The obvious predictions given are very likely to happen at some point in the future, but the insight here is educated conjectures that may or may not happen at one point in the future. This insight is probably given progress seen up until today. Both the obvious predictions and anticipated insight given are based on years of research, observation, and experimentation with applications, articles, and program source codes. Due to the existence of these obvious predictions and anticipated insight, the future of Computer graphics is foreseeable.

Works Cited

Belcher, Jim. "The evolution of computer displays." *Ars Technica*. N.p., n.d. Web. 29 Jan. 2012. <http://arstechnica.com/gadgets/news/2011/01/the-evolution-of-computer-displaysthe-evolution-of-computer-displays.ars/1>.

Bell, Nathan, Yizhou Yu, and Peter J. Mucha. "Particle-Based Simulation of Granular Materials." *College of Arts and Sciences*. U of North Carolina, n.d. Web. 28 June 2013. <http://www.amath.unc.edu/Faculty/mucha/Reprints/SCAgranular.pdf>.

Carlson, Wayne. "A Critical History of Computer graphics and Animation." *Design.osu.edu*. Ohio State U, 2003. Web. 13 May 2012. <http://design.osu.edu/carlson/history/lesson19.html>.

Daly, Mark. "The Future of Computer graphics: An Interview with Nvidia's Mark Daly." Interview by Kirk Kroeker. *Technology News*. TechNewsWorld, 10 Dec. 2003. Web. 5 June 2012. <http://www.technewsworld.com/story/32355.html>.

Fieggen, Ian. "Computer graphics File Format." *Ian's Graphics Site*. N.p., 29 Aug. 2010. Web. 5 Feb. 2012. <http://www.fieggen.com/ian/g_formats.htm>.

"GLSL Tutorial Von Lighthouse3D." *Zach.in.tu-clausthal.de*. N.p., n.d. Web. 4 May 2012. <http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/>.

Gorenfeld, Louis. "Lou's Pseudo 3d Page." *Extent of the Jam*. N.p., 3 May 2013. Web. 28 July 2013. <http://www.extentofthejam.com/pseudo/#basics>.

"Graphics & Games." *Computer History Museum*. Computer History Museum, 2006. Web. 5 Feb. 2012. <http://www.computerhistory.org/timeline/?category=gg>.

Hodgins, Jessica K. "Ray Tracing." *Computer Science Department at CMU*. Carnegie Mellon U, n.d. Web. 10 July 2013. <http://www.cs.cmu.edu/~jkh/462_s07/13_raycasting.pdf>.

Hruska, Joel. "Investigating Ray Tracing, the Next Big Thing in Gaming Graphics." *Extreme Tech*. N.p., 25 Sept. 2012. Web. 28 July 2013. <http://www.extremetech.com/gaming/135788-investigating-ray-tracing-the-next-big-thing-in-gaming-graphics>.

Huang, Jian. "Shader Programming." *The University of Tennessee*. U of Tennessee, n.d. Web. 26 June 2013. <http://web.eecs.utk.edu/~huangj/CS594F03/shaders/Shader_Programming.pdf>.

Macey, Jon. MSc Computer Animation and Visual Effects, University of Bournemouth. *nccastaff.bournemouth.ac.uk*. Web. 3 May 2013. <http://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/RayTracing4up.pdf>.

Mason, Matt. "6. Representing Rotation: Mechanics of Manipulation." Carnegie Mellon. Carnegie Mellon University. *Rensselaer Computer Science*. Web. 30 May 2013. <http://www.cs.rpi.edu/~trink/Courses/RobotManipulation/lectures/lecture6.pdf>.

McClanahan, Chris. "History and Evolution of GPU Architecture." *Mcclanahoochie.com*. Georgia Tech: Coll of Computing, 2010. Web. 4 May 2012. <http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf>.

McKesson, Jason L. "Learning Modern 3D Graphics Programming." *Learning Modern 3D Graphics Programming*. N.p., n.d. Web. 30 May 2013. <http://www.arcsynthesis.org/gltut/>.

Meiri, Etay. *Modern OpenGL Tutorials*. N.p., n.d. Web. 30 May 2013. <http://ogldev.atspace.co.uk/>.

Owen, Scott. "Computer graphics Hardware Overview." .*siggraph.org*. N.p., 1 June 1999. Web. 22 Apr. 2012. <http://www.siggraph.org/education/materials/HyperGraph/hardware/hardware.htm>.

"Pioneer Anomaly Solved by 1970s Computer graphics Technique." *Technology Review*. MIT, 31 Mar. 2011. Web. 29 Apr. 2012. <http://www.technologyreview.com/blog/arxiv/26589/>.

Premo□e, Simon, et al. "Particle-Based Simulation of Fluids." *Computer Science Department*. U of Utah, n.d. Web. 20 July 2013. <http://www.sci.utah.edu/~tolga/pubs/ParticleFluidsHiRes.pdf>.

Rademacher, Paul. "Ray Tracing: Graphics for the Masses." *University of North Carolina Computer Science Department*. U of North Carolina, n.d. Web. 20 June 2013. <http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html>.

Ramamoorthi, Ravi. "Online Lecture 3: Transformations 1 Basic 2D Transforms." University of California, Berkeley. *edx*. Web. 30 May 2013. <https://courses.edx.org/c4x/BerkeleyX/CS184.1x/asset/slides_transforms1.pdf>.

Ramamoorthi, Ravi. "Online Lecture 9: Ray Tracing 1: History and Basic Ray Casting." University of California, Berkeley. *edx*. Web. 30 May 2013. <https://courses.edx.org/c4x/BerkeleyX/CS184.1x/asset/slides_raytrace1.pdf>.

Ramamoorthi, Ravi. "Online Lecture 6: OpenGL 1." University of California, Berkeley. *edx*. Web. 30 May 2013. <https://courses.edx.org/c4x/BerkeleyX/CS184.1x/asset/slides_opengl1.pdf>.

Seidelin, Jacob. "Creating Pseudo 3D Games with HTML 5 Canvas and Raycasting." *DEV.OPERA*. N.p., n.d. Web. 28 June 2013. <http://dev.opera.com/articles/view/creating-pseudo-3d-games-with-html-5-can-1/>.

Sevo, Daniel. *The Future of Computer graphics*. Daniel Sevo, 2005. Web. 13 May 2012. <http://hem.passagen.se/des/hocg/hocg_1960.htm>.

"Shaderific - OpenGL ES development with GLSL." *iTunes*. Apple, 13 May 2013. Web. 30 May 2013. <https://itunes.apple.com/us/app/shaderific-opengl-es-development/id510588451?mt=8>.

Walker, Gianna. "The Inception of Computer graphics at the University of Utah 1960s - 1970s." *Silicon Valley Siggraph*. N.p., 27 Sept. 1994. Web. 22 Apr. 2012. <http://silicon-valley.siggraph.org/MeetingNotes/Utah.html>.

Wang, Yuan-Fang. "Lighting and Shading Models." *CS 180 Computer graphics*. U of California Santa Barbara, n.d. Web. 20 July 2013. <http://excelsior.cs.ucsb.edu/courses/cs180/notes/shading.pdf>.

Webster, Daniel. "History of Computer graphics 1970-1979." *Personal Webpage of Daniel Webster*. Daniel Webster, n.d. Web. 29 Apr. 2012. <http://www.danielsevo.com/hocg/hocg_1970.htm>.

"What is Computer graphics?" *Graphics.cornell.edu*. Cornell U Program of Computer graphics, n.d. Web. 15 Apr. 1998. <http://www.graphics.cornell.edu/online/tutorial/>.