University of Southern Queensland

Faculty of Engineering & Surveying

Next-generation 3D Graphics Engine Design

A dissertation submitted by

J. Cameron

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering and Bachelor of Information Technology

Submitted: November, 2006

Abstract

3D engines, software used to render images from 3D data, for the most part vary little in the technology utilised. All engines commercially available employ the same basic techniques for rasterisation. An alternative approach to that commonly used will be presented, which provides a cleaner result with comparable efficiency.

The history of 3D engines dates back before the era of personal computers. With such limited resources of the time, only basic routines could be used. As technology improved, these same methods were improved to eventually form what we now use today. While some of the modern adaptations are innovative, it is put forward that a change in technique from the standard practice is warranted to be more inline with the demanding requirements of modern applications.

Currently, all engines use polygons. Polygons consist of 3 points in space, which are joined together by line segments. A plane which passes through all 3 points is then clipped by these lines to form a triangle. Several alternative approaches to this ageing technology were analysed to determine viability on current hardware. Research was limited to approaches which do not use polygons. Several promising methods were found:

- Scanline rendering of curves
- Forward differencing of curves
- '3D pixels'
- Micropolygons

From these alternatives, it was determined the most viable was forward differencing of curves.

Reasonable analysis into this area is undertaken, along with provision for analysis of existing research on the topic. The use of bicubic patches is a core component, particularly using Bezier curves. For this reason analysis into such curves is also provided. While significant work is required to fully determine its effectiveness, it can now be seen that such an alternative technique to current 3D engine architecture is most certainly viable.

University of Southern Queensland

Faculty of Engineering and Surveying

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Prof R Smith

Dean

Faculty of Engineering and Surveying

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

J. Cameron

0011221117

_____

Signature

_____

Date

Acknowledgments

I would like to acknowledge those that helped me with suggestions and also those that supported me through this more stressful time. I would first like to mention my saviour Jesus for giving me the ability to do this project despite all the circumstances. I would like to thank Dr. John Leis for all the input he's provided me over the whole course of the dissertation, in particular when shipment difficulties arose and when I was sick. I would also like to thank my family for supporting me and arranging me to go on retreat to get the bulk of the programming done. I would also like to thank my friends that were very understanding when it came to the workload, and provided me with guidance on how to approach my studies. Thanks guys.

J. Cameron

University of Southern Queensland

November 2006

Contents

Nomenclature

3D Acceleration	Use of specialised hardware to increase the rendering speed of a 3D engine. Requires special access such as an hardware-accelerated API.
3D Engine	A software system which takes raw model data and produces an image or series of images.
6-neighbours	Adjacent voxels which share a face
6-connected path	A path of adjoining voxels in which each voxel is connected to the next via one of its neighbours' faces
18-neighbours	Adjacent voxels which share at least one edge
18-connected path	A path of adjoining voxels in which each voxel is connected to the next via at least one of its neighbours' edges
26-neighbours	Adjacent voxels which share at least one vertex/corner
26-connected path	A path of adjoining voxels in which each voxel is connected to the next via at least one of its neighbours' vertices/corners
API	Short for Application Programming Interface. A set of libraries from which programmers can quickly create an application using its pre-built routines for a specific purpose.
Backface Culling	The process of removing the reverse side of a surface (traditionally a polygon).
Bézier Curve	A special but common representation of a curve that is depicted by 4 control points. Used for examples that model free-form curves and surfaces.

Bicubic Patch	A curved surface represented by the combination of 4 curves arranged along its edges.
Clipping	The process of slicing an object in order to remove a section. Often used at boundaries. Has the Boolean operation equivalent of an OR.
Collision Detection	A means of determining whether an object lies within another object. Also traditionally used to determine whether an object is moving through another object.
Cubic Curve	A curve which can be represented by a cubic function, but does not necessarily have to be in that form. It will thus have 4 coefficients.
Cull	To remove objects from consideration, typically for falling outside a designated area.
Depth Buffer	A section of memory used to store candidate pixels prior to rendering. Depth information is also stored so subsequent candidates for the same pixel can be compared and judged according to distance from view.
Dicing	The process of generating micropolygons from another representation.
DirectX	A 3D graphics API developed by Microsoft to compete with OpenGL. It is now the most-used such API.
Distance Field	The area between the Hither and Yon region in which models can be seen. Objects outside these distances are typically culled.
Eye Space	The coordinate system determined by an objects position relative to the 'eye', or viewing point. Similar to screen space.
Forward Differencing	An alternative to the traditional polygon method in which curves are stepped through for at least every pixel drawn with moderate speed.

Frame	The current image to be drawn, even if it does not contain all elements at the time. Can also refer to an image that has been rendered or will be rendered as part of a series of renderings (or 'Frames').
Frame Buffer	The memory that has been allocated to contain the current or next frame. All current implementations use a frame buffer, however that has not been historically true.
Height Map	An image in which all pixels are instead references to the height of the pixel with respect to the plane upon which it has been placed. The use of one is usually restricted to offline renderings, as the high calculation process is especially slow.
Hither	A term used to indicate the closest an object can come before clipping or culling.
Hybrid Engine	An engine that employs multiple, unrelated ideas in its implementation.
IrisGL	The proprietary predecessor to OpenGL owned by SGI.
LGPL	A licensing agreement used commonly by open-source products which allows for modification and use without special permission.
Micropolygons	Alternatives to regular polygons. Sub-pixel size and fast rendering make them a candidate for an alternative 3D engine design.
Mipmapping	The process of storing smaller versions of utilised textures in order to improve rendering speed with seamless integration. Can be used in any situation where texturing is required.
Object Space	The coordinate system relative to the object in question.
OpenGL	The first true standard 3D engine API. Spawned from SGI's IrisGL, it is now overseen by an independent board.
Open-source	Software with a license that includes the source code (the code the software was written from) and permission to modify this code. Is common in software that is intended to be standard.

Pixel	A single 'dot' on the computer screen. With the viewable area of an image being divided into a matrix of small cells, this would be one cell.
Rasterisation	The process of generating pixels.
Ray Tracing	A slow but accurate and visually appealing method of rendering objects. Particularly used when accurate shadows or lighting is needed.
Render	The process of converting data from the scene into an image.
Scanline	A row of adjacent pixels having the same x coordinate.
Scene	The entire set of objects to be rendered.
Screen Space	The coordinate system based around the position relative to the screen. Similar to eye space.
SDL	An API used for generating both 2D and 3D graphics. Includes much of the OpenGL API as well as input functions.
Slicing	The process of splitting a surface into multiple smaller surfaces.
Tessellation	The process of converting an object into polygons from any form.
Texture	A matrix of pixels used to represent an image that will be overlaid onto a surface. Is included in all currently used 3D engines.
View Space	See screen space.
Voxel	A '3D pixel'. A cell or a 3-dimensional matrix used to represent an object. Traditionally also used for various other things.
Wireframe	The representation of an object that includes only the outline of its surfaces. Traditionally represents only the edges of an objects' polygons.
World Space	The absolute coordinate system used in a scene. Commonly used to reference objects contained in the scene.
Yon	A term used to indicate the farthest an object can be before clipping or culling.

Chapter 1

Introduction

1.1 3D Graphics Engines

At the heart of any program that displays 3D graphics is a 3D graphics engine. A 3D graphics engine is a piece of software that transforms 3D object data into an image. This data can be in any form, and is usually tailored towards the engine in question. Often software development companies produce a piece of software that just contains a 3D engine, and other software development companies use these engines to make programs. Standard mechanisms have been developed to improve 3D Engine creation, such as OpenGL and DirectX. Almost all computers nowadays include a hardware implementation of both systems.

Due to the complexity and vastness of the data to be represented, the 3D engine is always the bottleneck of any program in which it is used. Normally it is the component that takes the longest to create. A vast amount of research has been undertaken to make existing 3D engines faster. Both special hardware and software exist in order to improve performance. People pay top dollar to obtain the latest, fastest 3D acceleration hardware in order to make the 3D engine they use faster. This translates to a large amount of money and a large market.

With all this increase in speed, 3D engines have managed to be fast and contain an ex-

traordinarily vast amount of detail. What I perceive is missing, however, is smoothness. This translates to accuracy in the models being represented. All current 3D engines in use today utilise the same method for displaying images. While it is fast and (relatively) simple, it is not accurate (smooth), and produces obvious visual aliases. These are in the form of 'blockiness' that can been seen around the visual edge, or silhouette, of objects. The intention here is to provide an alternative 3D graphics display engine that will not display such visual aliases, yet still be competitive in speed.

1.2 Brief Overview of the Dissertation

Presented in this paper is four reasonable candidates for consideration for being a next-generation 3D graphics engine. The functions, strengths and weaknesses of each of these is examined, and a suitable candidate chosen. Further analysis into this chosen alternative to current engines is undertaken to find additional optimisations and techniques. Lastly, an implementation (of sorts) is presented which is designed to testify to its ability. Brief analysis is performed and a conclusion is reached.

Chapter 2

A History of 3D Engines

2.1 Polygonal Engines

The idea of 3D graphics (as we know it today), and thus the idea of 3D graphics engines, has been around since mid-century. At that time computers were in their infancy. Computers of the time spanned floors of a building, and were extremely expensive. 3D graphics engines were initially thus built on very limited hardware. They consisted of points in world space (absolute cartesian position) that are transformed into eye space (position relative to the view). These transformations involve trigonometric operations, which are not fast to process. Once in the correct position, lines are drawn between these points, creating edges. The end result, called 'wireframe', took on the typical form of a series of boxes loosely resembling simple objects.

Even drawing a simple line took a fair amount of resources for the technology of the time. Thus the first 3D graphics engines were totally line-based. While such methods were an achievement for back then, it is still very basic compared to what we have now. The first optimisations were thus optimisation to the line-drawing algorithm. Most notably the Bresenham line-drawing algorithm was both the first and the most popular way of efficiently drawing lines. Others later followed that improved on it, but these came too late to be noticed.

The trouble with this representation was that the objects were essentially see-through. The lines represented important edges of the objects, but objects behind them could be seen as though they were in front. This caused a problem where both the position and orientation of the objects were often vague. To overcome this, the idea of a solid object was introduced. The gap between the lines was quartioned off, and nothing behind the object was rendered (drawn to screen).

This gave birth to the idea of Polygons. These quartioned off areas could now be considered flat surfaces. Polygons are are thus planar polyhedrons that are represented in space as 3 or more points. Lines, visible or otherwise, are drawn between all these points in a particular order and the space between these lines is filled. Nowadays most polygons consist of 3 points to avoid ambiguity in the event that all points don't lie flat on a plane. Polygons are simple and very fast to render, which makes them very convenient to implement.

2.1.1 Evolution of Polygons

As time progressed, the technology improved. Polygons were filled with colour rather than being left blank. This created a more 'solid' look. Also by this time, colour monitors were commonplace. The concept of lighting was then introduced. This took the form of Polygons' colours being reduced based on the angle they made with a light source. This made them appear much more like what they are today.

The difference between different polygons was still considerably obvious, so better shading was introduced to make the whole object congruent. This is achieved by calculating the normal vector (tangent vector to the plane - used to calculate the angle the plane makes with the world and with lighting) for each vertex rather than for each polygon. Blending of light intensity between the vertices of a polygon ensures congruence. This is especially so as most polygons share vertices to form surfaces. Of course, like every step, the engine got slower. While most other features' calculation costs are evened out with the increase in hardware performance, it is not the case here. Almost all real-time 3D engines in use today utilise the older affore mentioned shading method, however this is expected to change in the future (around 40 years after it was developed).

Regardless of the shading policy, polygons still only contained one colour. While colour blending between vertices in much the same way as light blending would help, considerably more is needed to create a believable result. Textures thus came into being. Textures are 2D rasterised images used for overlaying onto polygons. The vertices then contain information about their position on the image. Upon rendering, the points in-between the vertices are interpolated based on these co-ordinates. Once textures are placed on a polygon, the object then starts to resemble what it is supposed to be. Of course, as with everything else, the texture position on a polygon is only truly accurate at the vertices for curved surfaces unless it is perfectly rectangular.

At the time, textures were only small images, around 16×16 pixels. This was due to the memory limitations of the systems of the time, the early ones having to use ROM or magnetic media. Individual texture pixels were very obvious, as they usually took up multiple screen pixels and produces a grid-like appearance when viewed up close. The blockiness of objects in general was also still very much apparent. As technology improved, the appearance of produced images improved as well. More polygons were able to be included while maintaining acceptable speed. Textures got larger as well. Eventually hardware was able to blend neighbouring texture pixels together for screen pixels that fell in-between two texture pixels. This dramatically improved appearance, but was not feaseable in any software implementation.

The process is very slow, however, for viewing large textures at a great distance (essentially shrunk). Thus mipmapping was introduced. Mipmapping is a process where successively smaller versions of a texture are also included (or generated). These sub-images are used instead of the full image for cases where a large proportion of texture pixels are skipped (i.e. at long distances). Further development allowed the technique to even be used for many levels within the one polygon. 'Tearing' (a line of noticeable change) was visible as the texture was changed from one scale to another, however it was not too obvious and was worth the speedup.

To increase speed, lighting was calculated prior to use in real-time applications, typically at model compliation time. This provided a convenient way of producing better-looking images at a lower cost (to the renderer). In such circumstances, a crude form of shadowing was also possible. Various light points (ray traces of sorts) were taken at

regular intervals, and mapped onto surfaces. Due to the added rendering load, these points were fairly sparse. Of course, this only worked on occasions where the models would not move in such a way as to conflict with the expected lighting.

Models which are stationary to either the view or the world were feasible, however everything else was not. This problem was solved by categorising models into world, moving and view. World models were those that are always fixed relative to the global x, y and z co-ordinates. Decent lighting and shadowing was quite feasible for such models. Moving objects raised the most problems, as they could be oriented in any direction. Mid-range lighting was placed in strategic points in order to attempt to blend in with the surroundings, and lighting was then calculated as before. Nowadays this is all done at rendering time.

2.1.2 Hardware Implementation

In the early 90's, specialised hardware began to emerge. By this time, 3D engines had progressed far enough that simple forms were able to be implemented for real-time applications. When 3D games emerged, the market for all things 3D opened up, and now high-volume production of hardware was possible. At the start, it took the form of an expansion card, which pluggs into a standard expansion port in a PC. The video card then blugs into this card, as does the monitor. This specialised card was called a 3D accelerator, and was particularly designed for games. The games of the time were designed for ordinary PCs, however they had special extensions to allow for '3D acceleration '.

Just prior to this, when 3D acceleration cards were quite expensive and designed specifically for high-end machines, various APIs were introduced. Large firms that manufacture specialised minicomputers and microcomputers developed libraries and systems that forms the basis for efficiently implementing 3D engines (that uses polygons). This was done for specific hardware, and had drawbacks when trying to implement engines that supported different hardware. Even when implementing for different hardware from the same vendor, programmers had to manage features that existed on some systems and not on others. Some of these libraries even grew into APIs (Applica-

tion Programming Interfaces). This means that it is then possible for engines to be implemented in different programming languages.

The most prominent of the companies to do this is SGI. SGI developed a popular framework called IrisGL. It was significanly easier to use than the standard of the time (PHIGS), and gained popularity. Eventually it became apparent that due to large competition between companies, any API developed would be meaningless in the push to make programs compatible with hardware from competing vendors. Programmers refused to go with just one company, and therefore did not use any proprietary API. Being robust enough, SGI decided to release IrisGL to the community as a standard API.

'Releasing to the community' means to make it open-source. This means that control of the code is no longer in the hands of the originating company and never will be. Anyone can look at the underlying code and even change it, given a few loose conditions. This code thus had to undergo significant change in order for it to be read and accepted. Along with that, in order for it's growth to be controlled, an independent body comprising of representatives from the industry was formed. This body's sole purpose is to ensure it gets used in an effective and innovative manner. SGI also took this opportunity to revise the API before release, enough to warrant a new name, OpenGL.

OpenGL quickly became the standard interface, however hardware took a long time to catch up with it. A new feature was that if hardware is unable to support all the features of OpenGL, these features would be implemented in software. Even with this, budget hardware, like that found in a PC of the time, was still not powerful enough to take the load required for a smooth experience. Thus subsets of OpenGL were common, such as Glide. These subsets behaved like the fully-fledged OpenGL, but with many of the instructions unavailable. Hardware eventually grew enough to support the entire API, however by then subsequent revisions included extra features that were beyond this. In this way, OpenGL has always kept ahead of budget, home-user hardware, making engines run in an awkward hardware/software conglomerate.

A few years after OpenGL was released and had gained popularity, Microsoft released

DirectX, which runs purely under Windows. This is a rival API designed to compete with OpenGL. DirectX works in a similar manner to OpenGL, however it includes features outside the scope of 3D graphics including user input control, windowing control and 2D graphics. Like OpenGL, DirectX also suffers from a lag in vendor hardware implementation, resulting in a mix of hardware acceleration and software emulation.

2.2 Need for Change

Thus current 3D graphics engines have evolved considerably. 50 years of modifications and implementations have made them what they are today. Most consider a 3D graphics engine and a 3D polygonal engine one and the same. Due to competition by rival companies and standards, 3D acceleration hardware has progressed massively. Most of the 'legwork' for an engine has already been done by the API, so the programmer now no longer has to be concerned about the mechanisms 3D egnines utilise. Along with that, the basic knowledge of how a 3D engine is waning out of existence. Programmers almost always use one of the two standard APIs, and those that don't still use traditional architectures designed by someone else.

In short, it's simply too convenient to use current technology to design a 3D application. Numerous tools exist. Due to the large amount of support, I doubt if most even think about whether there is an alternative. The culture is that 'if it aint broke, don't fix it'. While this may prove to work, it doesn't mean it's necessarily the best implementation possible.

Also due to the strong advances in hardware, it is viewed that any alternative would be significantly slower due to the lack of acceleration. This paper will not use any special features of 3D acceleration hardware. With this said, that does not mean that implementations need not use such features. OpenGL's relatively new 2.0 and later specifications were numbered so due to the addition of a new programming specification. With this, programmers can now develop small routines that are purely run in hardware (assuming the hardware supports it – which it will, as discussed prior). It would be reasonable to assume that soon any given alternative 3D graphics engine im-

plementation would have a major component that can be implemented in this. This will effively '3D accelerate' the purely software implementation. Such methods, however, are beyond the scope of this research.

To recap, the slow evolution of the traditional 3D engine model has made it what it is today. While it is due to hardware constraints forcing the prior implementations down a particular path, this path need not necessarily be the fastest or most efficient. Most consider it the best because current trends and technology favor this method, and indeed most perceive it as the sole form of 3D engine. This is unnecessary, and untrue. This paper strongly suggests that technology has evolved enough to be able to support a new, better technique for generating 3D images, that sheds away the long history of slight modifications from basic principles. To be consise, a next generation of 3D engine is now plausable and warranted.

Chapter 3

Curves and Curved Surfaces

3.1 Curves

The main problem with current 3D engines today, as stated before, is the visual artifacts created by using sparsely separated vertex points joined together by linear patches. A linear approximation to a surface is simply not good enough for most objects. Any object that has a curved surface is only accurately approximated around the vertex points. All other areas display a huge deviation from the shape intended. One option is to increase the vertex count to reduce the error. The more vertices, however, the slower the image renders. Nowadays this is OK for offline rendering, where the speed is not important. Applications such as CGI for movies or still images take advantage of this technique. For the majority of applications, however, real-time rendering is involved. There is a limit to the number of vertices and polygons that can be accommodated before the usability of the entire program is compromised.

Thus an alternative method for representing image data is warranted. There are various methods for representing objects, however to design accurate models, certain techniques are necessary. While some of these techniques generate data that is not in any curved form, means of designing models is necessary in order to get this data. Thus, curved object representation is necessary.

Regardless of which representation of curves you use, it can always be transformed into a power series curve. These curves take the forms such as equations 3.1, 3.2 and 3.3.

$$x = ay + b \tag{3.1}$$

$$x = ay^2 + by + c \tag{3.2}$$

$$x = ay^3 + by^2 + cy + d \tag{3.3}$$

Equation 3.1, for instance, represents a line, regardless of the form used to render it. This could be classified as a 1^{st}-order curve (even though it does not curve at all). A 2^{nd}-order curve, referred to a quadratic, is represented by equation 3.2. A cubic, a 3^{rd}-order curve as represented by equation 3.3, is the lowest order possible in order to approximate arbitrary curves. Thus cubic and higher curves are what will be employed to generate more accurate images.

Various means of curve representation and rasterisation are presented here as some approaches use this directly to generate images. Other schemes are also available that can be either used to replace these curve representations or to generate models that can be used in certain implementations.

Bézier curves are by far the most commonly used curves for research into computer graphics. Originally created by Pierre Etienne Bézier for use at Renault, it is one of the oldest and most convenient representations around. Bézier curves consist of 4 points. 2 are the endpoints of the curve, and the other 2 define how much the curve bends, and towards where. Bézier curves still follow the formal rules for curves, though. Having 4 points means it's a 3^{rd}-order curve. It can be translated into normal power representation easily.

The advantage of Bézier curves is they are easily deformed by moving the control points, and are also easily converted to power representation. Given the widespread use of Bézier curves, I am not surprised that they are the chosen method by all alternative 3D engines discussed in this paper.

Chapter 4

Alternatives Engines

4.1 Overview of Alternative Approaches

Many alternatives are presented here. Selection is based primarily on the ability to render images that are more accurate than that produced by polygonal engines and with comparable speed. Due to the complexity of creating higher precision engines, it is obvious that a low-polygon traditional 3D engine will be faster. Polygonal engines have the advantage of being able to sacrifice image quality (and thus precision) in favour of speed. The goal is thus to choose an approach that has a high likelihood of improved performance over the polygonal engine of similar quality.

4.2 Scan-line Rendering

Scan-line rendering works in a much similar way to the current methods used for polygons. With polygonal methods, the common (and fastest) way of generating pixels is to join polygon points up with lines and sort them by y, then x. A 'scan-line' is then introduced to scan the y direction from top to bottom. For each (horizontal) line, points between the two edges of that section of the polygon are filled. (Whitted 1978) was the first to produce such an implimentation. At the time, disk access was the major bottleneck in for hardware. Implementations as was common back then used a

frame buffer to store rendered pieces of the image before everything was rendered. This caused severe trashing of the harddisk. The scanline method did no do this however. By calculating it in scan-line order, sequential disk access was possible. The meant that little movement in the heads of the harddisk was necessary and that no time was needed for the disk to move to the right orientation.

A significant speedup would have resulted from this method. It thus spawned everal other scan-line techniques. Nowadays this is not an issue. Rendering is performed in RAM, which has the same access time regardless of the memory's location. A framebuffer poses no problem then. In fact, a depth buffer is employed to distinguish two objects that occupy the same x space.

A while later, (Blinn 1978) decided to re-explore this idea. It is unknown why it was never taken up, however Blinn's earlier work focused on shading techniques for such surfaces. In face, Blinn is quite well known for this. Blinn's algorithm, however, quite closely resembles Whitted's. Even the diagrams are familiar. He does go into more detail about the different situations that can cause problems in such an implementation though. In particular, when there's a local maximum or minimum in one dimension, or a combination of the two, the algorithm has to take special care in order to produce the correct result. Along with the other sections which are described further down, a special routine has to be implemented to correctly handle such situations.

(Lane, Carpenter, Whitted & Blinn 1980) looks at different scan-line algorithms including those put forward by Whitted and Blinn. The lane-carpenter method is also presented, however this is a hybrid engine that also uses polygons. Here we see how scan-lines can be used to create polygons for use with traditional engines.

For each of these, the program essentially scans through each y line from top to bottom, generating pixels for each polygon from left to right. It can be thought of as a series of planes cutting through the z and x axes at each y pixel increment. This is reasonably straightforward for linear polygons, as lines are reasonably fast to calculate (especially considering a large amount of research is being done to make them even faster). Lines are also monotonic in all axes, meaning that for any given x, y or z value, there is at most one pixel value for the line.

Such luxuries do not exist for curves, however. Attempting to implement a similar technique, Whitted developed a method for scan-line rendering of cubic curves. He decided to use Bézier curves as his curve representation form. The first problem he encountered was the existence of a potential silhouette in surfaces. A special pre-processing algorithm had to be implemented in order to find all silhouettes. A problem which arose with that was that the order of silhouettes was not limited to 3^{rd}-order (cubic) curves. Higher-order curves were produced, and thus needed to either be sliced up or approximated to fit the rendering algorithm.

Another problem that arose was that since the curves have the potential to be neither monotonic in x or y, they had to be split up in order to fit the algorithm. It's designed in a similar way to traditional linear algorithms, and thus only works if the curve does not curve in on itself. Considering curves are being split anyway, silhouette curves are also approached likewise. Thus, extensive pre-processing is done to ensure all the curves are in the right form.

Once that is achieved, (Whitted 1978) uses Newton's iteration is used to calculate the x values. This process is made more efficient by an estimate being taken from the previous solution, and iterated. Occasionally a solution is not found, however, and more expensive procedures have to be taken to find the point.

(Blinn 1978), however, uses heuristics and numerical techniques instead. This can work much more efficiently in a lot of cases, but has a major drawback. (Lane et al. 1980) mentions that this method may fail, and more often than Whitted's method. If that is the case, as before, expensive, brute-force methods have to be used to resolve the situation.

This method seems to be solid, and backed up by other research. Scan-line rendering has the advantage of a speed boost due to the successive writing of pixels to adjacent locations. This, of course, due to hardware advances, does not produce as much of a speed boost as it once did. The rendering of pixels between curves is also very fast. What lets it down is the curve calculation. Newton's iteration is fairly slow, especially when it fails and you have to go by brute force. A lot of calculation is involved pre-rendering, as local maxima and minima have to be found along with silhouettes. All

these so as to slice curves up appropriately (and for every frame).

4.3 Forward Differencing

Forward Differencing uses a completely different technique. Here, curves are start off initially in the Bézier form. At each frame, they are converted into parameter space. This is another form which has a direct correlation to the Bézier form. Thus they can easily be converted back if required.

(Klassen 1991*b*) intoduces the method well. A large amount of detail goes into explain how Bézier curves are used. It can be seen that any curves can be used (although any less than cubic would not be suitable for arbitrary surfaces), however most of the research indicates a strong preference towards Bézier curves. Subsequent research also keeps to this ideal.

In parameter space, functions are monotonic in one value, t, which does not represent any axis. Thus a curve that curves in on itself in a normal axis can still be expressed as a function, as it does not do so in parameter space. One significance of this form is that 3 equations are produced; one for each axis. While the t does not represent any particular axis, the result of stepping through t is the axis location for points along the axis in question. Thus to render a curve, t is stepped through from one value to another, with the same values for t used for every axis.

The stepping process is not an easy one. With traditional methods using cubic function directly (or derivations thereof), a large number of multiplications are involved. For example, the following cubic function for the x axis, equation 4.1, is expressed in the computer as equation 4.2.

$$x = 4t^3 + 3t^2 + 2t + 1 \tag{4.1}$$

$$x = 4{\times}t{\times}t{\times}t + 3{\times}t{\times}t + 2{\times}t + 1 \tag{4.2}$$

While the number of multiplications can be halved, each operation still takes a proportionally large amount of time to compute compared to an addition. Multiplication and division operations are very computationally expensive, and thus must be avoided. Forward differencing uses little of these operations, and usually uses bit shifts instead of such operations. This will be further discussed in the next chapter.

(Shantz & Chang 1988), (Klassen 1991*a*), (Klassen 1991*b*) and (Klassen 1994) all deal in the same area. They detail explicitly how forward differencing works, and some provide extra information on how it may be improved. They do bring forward different implementations, though, and some often provide different solutions to inherent problems. Thus, to analyse and explain how forward differencing works, all four papers need to be considered at once. A point to note now is that all subsequent formulae, etc. come from these sources — usually all of them.

Curves, of any form, first need to be converted to the correct form. The only valid conversion from Bézier is to power form. This is done in the usual manner. Forward difference form is related to power form by equation 4.3, which can be derived from the general form found in (Shantz & Chang 1988) (which has a typographic error, specifying $d+1$ instead of $d-1$). The conversion from power form to forward difference form is done using the matrix in equation 4.4 from the same passage.

$$a_0 + a_1 t + a_2 t^2 + a_3 t^3 = b_0 + b_1 t + b_2 \frac{t(t-1)}{2} + b_3 \frac{t(t-1)(t-2)}{6} \tag{4.3}$$

$$FD = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 6 \\ 0 & 0 & 0 & 6 \end{bmatrix} \tag{4.4}$$

Once in this form, the curve can then be 'stepped through'. The forward matrix, denoted in equation 4.5, also derived from (Shantz & Chang 1988). This simple operation masks a more complex theory behind it. As stated before, the curves are a 3^{rd}-degree polynomial. We start at one end, which is known already. If we differentiate this curve, we get a 2^{nd}-degree polynomial. A successive differentiation yields a 1^{st}-degree polyno-

mial. This becomes a linear equation of which the slope is trivial to find. Using a set distance, we can trace this slope to the next point. Once obtained, the new point found is treated as the original point in a new curve that resembles the original curve but is shifted up the step distance. Earlier representations used a different matrix which too more time to equate. This was using a slightly different basis, and once it was found that the basis could be converted, this better representation was found.

$$E = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

In order to resolve the problem the scanline method has, a final change in representation is warranted. In this case, t is stepped from 0 to 1. This is a totally stable region in which the calculation will always produce the correct result and not default. This speeds up the process greatly.

While t is stepped through with even steps, it does not hold that the steps will be even for any axis. This means that even though for some sections of the curve every pixel may be filled using a certain step size, other parts of the curve can produce gaps for the same step size. Moreover, some parts of the same curve may have the pixel filled numerous times, resulting in a huge reduction in efficiency. Thus, in order to ensure that all pixels are filled, a relatively small step size has to be chosen, which can result in a large number of points falling on the same pixel. This endangers the viability of the procedure, as too many operations could be performed for each pixel.

To alleviate this, several methods have been put forward: adaptive subdivision and adaptive forward differencing. Adaptive subdivision also solves the problem of accuracy by splitting curves (subdividing) prior to forwared differencing. Adaptive forward differencing adaptively changes the step size used depending on whether there was a redundant pixel or a pixel gap in the rendering. More detail into these methods will be explained in the next chapter.

One other important thing to note with forward differencing is that errors accumulate.

Due to each step producing a new curve, any error carries on for the rest of the curve. Thus, if a large number of points along the curve are found (which would be not uncommon), the error may grow large enough to be noticeable.

4.4 '3D Pixels'

'3D Pixels' is a term colloquially used to describe a number of 3D engines developed around a concept of a cubic frame buffer. The term 'Voxels' was traditionally used, however it is now used to describe several newer components used by traditional 3D engines. This term originated from the term 'volumetric pixels', and over time evolved into several varying ideas. I will use the term 'Voxel' to describe a single 3D pixel, as the term 'pixel' by itself usually implies a 2D pixel. There are several flavours of '3D Pixels' including CUBE, GODPA, PARCUM and $3DP^4$ architectures. My analysis is on the most recent and the most comprehensive form, the CUBE architecture presented by (Kaufman 1987).

'3D Pixels' are centred around the idea that 2D pixels can be extended into 3D. Pixels are simply points on the screen, referenced by the horizontal and vertical (or x and y) position. Because of the cardinality of the pixels, they are seen as squares covering the screen. Pixels are organised into matrices in traditional 2D engines. These matrices are called 'sprites'. 2D engines use pixel mapping to move the sprites around the screen forming the basis for the engine.

This is extended precisely in 3D, however the operations get a bit more complicated. As well as movement, zooming and rotation have to be performed. For this, the size of the pixels as well as the transformation operation have to be considered. This method breaks away from the 2D model a bit more for this reason.

A voxel is obviously thus just like a pixel except for the extra z dimension. Thus, a 3D matrix of pixels is formed for every object. The biggest problem with this idea is that a large amount of memory is required to store everything. For example, the normal resolution for PCs is 1024×768 pixels. So, in order to accommodate an object that fills the screen and performs appropriate operations, the largest dimension must be taken to

all dimensions. The standard bit depth for each pixel in contemporary applications is 32 bits, or 4 bytes. This means, this object would be 1,024 × 1,024 × 1,024 × 4 bytes, or 4GB (4,294,967,296 bytes)! This is more than the total RAM for most computers. Obviously this is unacceptable, and poses a formidable obstacle. It is quite possible to use a reduced resolution, however the quality of the result will be reduced.

100% perfect quality is not what we're looking for, luckily, so this approach still warrants consideration. As hinted previously, significant research has been done into this area, indicating the plausibility of a decent implementation.

Due to the fact that objects can be viewed at any angle, a method has to be developed to find the correct voxels to render into pixels. As the object may be further away, voxels will be missed. The CUBE architecture uses a traversal system to find the correct voxels. Starting at any arbitrary voxel (you could optimise it to find a nice match, however this would lower the performance slightly), specific algorithms are used to find the appropriate voxels for neighbouring pixels. These will be explained in general shortly.

Adjacent voxels are categorised into 3 categories: '6-neighbours', '18-neighbours' and '26-neighbours'. 6-neighbours are the voxels that share one face (thus the 6 faces of a cube). 18-neighbours are voxels that share one edge. They thus each share 1 face in 2 6-neighbours. 26-neighbours are voxels that share one vertex. They thus share 1 face in 3 18-neighbours and no faces in any 6-neighbours. So, for a change in 1 axis (for example to move 1 voxel in the x direction, and none in the y or z directions), a 6-neighbour is found. For change in 2 axes, a 18-neighbour is found. Likewise, for a change in 3 axes, a 26-neighbour is found. This concept lends itself to paths as well. A traversal between several 6-neighbours is called a '6-connected path'; between 18-neighbours, it's a '18-connected path' and between 26-neighbours, it's a '26-connected path'. In this manner, it is easier to find the next voxel to be considered.

(Kaufman 1987) mainly demonstrates several algorithms for efficiently converting other representations into voxels. This is also reflected in the later paper, (Kaufman & Shimony 1986). Here's the conversion practices is extended to various different implementations based on the object type. An interesting note is that a '3D Curves' object

type is presented with a very similar implementation to the forward differencing method previously discussed. Kaufman and Shimony have also extended this into surfaces, providing insight into how 3 dimensions could be implemented using this technique. Thus you could consider a hybrid engine where bicubic patches are converted into voxels using forward differencing and all the additions previously discussed (as well as some discussed later).

Once the connected paths are established, Kaufman assumes it is easy to obtain an implementation for rendering. Successive voxels are selected based on the difference between the x, y and z values. Thus, if most of the change is in one axis, a 6-connected path is used. Likewise, for 2 axes, an 18-connecte path, and for 3 axes, a 26-connected path. I can be demonstrated, however, that this is not the optimal method for voxel selection. L-shaped paths access the next voxel. A recommendation for improvement would be to move on a diagonal, using a differently-connected path until equilibrium is reach whereapon the correct path is used.

To provide an example, imagine how a knight moves on a chess board. It moves 2 squares in one axis, and 1 in the other axis. If you consider it this way, the knight has moved 3 squares. If, however, you consider it to move 1 square in a diagonal axis (i.e. 1 unit in 2 axes — a higher connected path) and then 1 square in 1 axes, you will yield the same result. The knight, however, has only moved 2 squares. This might only be a 1-square advantage, however it increased the efficiency of the algorithm by $\frac{1}{3}$. This is scalable linearly to an arbitrary number of squares, and is also scalable to the 3^{rd} dimension.

The advantage of both these methods is that pixel- and voxel-finding operations are reasonably quick. This is offset, however, if 'in-between' pixels (i.e. pixels that don't have a corresponding voxel due to the object being too close to the viewing screen) need to be found. Morphing or changing objects may pose a problem, as a large number of memory access operations (as well as numerous calculations) need to be performed in order to re-populate a matrix. Most simulations, however, require little of this. If memory if abundant, the option to pre-populate 'frame' matrices can also be considered. This is where an animation is pre-empted, and motion is drawn offline in numerous separate matrices. All that is required, then, is to change matrices for the

next 'frame'.

The most prominent benefit for using this method, however, is it's ability to display non-solid objects and objects that cannot be represented by curve patches. For non-solid objects like particle fields, smoke, clouds, etc. many voxels have to be accessed and rendered to a single pixel. This can be emulated by changing the value of the outside voxels to emulate this state (producing less voxels). Other non-solid objects such as crinkled plastic or cloth can be easily displayed using a similar technique. In fact, such objects, transparent or otherwise, lend themselves better to this representation than any other, as the complexity of the object being represented as no bearing on the rendering time. 3D engines, are more commonly found these days with such items, especially when dealing with character animation. Solving the problem of deformable material such as cloth and hair has long been seen as one of the most difficult and important obstacles facing 3D engines.

One last thing to note is that engines can be developed that use both this method, and other methods *at the same time*. This is not a hybrid engine as is mentioned previously. Here, voxels and polygons are used in the same scene without any conversion between the two. (Kreeger & Kaufman 1999) shows an example where a polygonal jet is flying through a voxel-based cloud. Two totally different implementations working together to produce a working image. This would be an area for further study.

4.5 Micropolygons

Much less research has been done into the area of micropolygons than the previous areas. That being said, it has been successfully used by companies like Pixar to render high-quality, offline scenes. Although it was not used for real-time purposes back then, 20 years have passed, and thus the method is worth considering.

Micropolygons are basically small polygons. They behave in a similar way to regular polygons, but the approach used to deal with them is vastly different. Micropolygons have a maximum size of $\frac{1}{2}$ pixel when converted into screen space, and therefore only ever affect one pixel. With anti-aliasing methods, they can affect more than one sub-

pixel though. Considering their size, they need only contain one colour. This makes texture mapping simpler and quicker than any of the other methods discussed.

All rendered objects are eventually converted into micropolygons for each frame. This is called 'dicing'. Going over the details, I cannot see why this has to be done so often. If objects change form or move dramatically (so that either some micropolygons are greater than $\frac{1}{2}$ the size of the pixel or they become too small to handle efficiently), then re-conversion needs to be performed, however most objects do not behave as such often. I propose that computation-saving efforts be brought forth in order to limit this process. Diced objects could be stored in separate buffers to be used on subsequent frames, saving a large amount of computation time. Considering the age of this method, perhaps the designers felt that too much memory would have been used (at the time, the maximum amount of RAM on offer would have been 1MB – 1,048,576 bytes). All model data would have been stored on a hard-drive, making memory access a severe bottleneck. This would have led to the designer's strong preference to limit memory thrashing, which ultimately affected the design choice.

History aside, the idea behind the concept still holds strong. (Kreeger & Kaufman 1999) lays it out in detail (and unlike the other alternatives, this has been used in movies). Objects are originally in any arbitrary form, of which a bicubic patch is recommended. The viewing area (an envellope for determening whether an object will be within view) is considered, and all objects not falling within this are removed. The distance field has also to be considered. Drawing from the traditional polygonal method for rendering, objects will undergo a perspective transformation, and thus be subject to mathematical instability if an object is too close to the 'eye'. Unlike traditional methods, however, objects are not immediately trimmed to make them either in-view or far enough away from the 'eye'. This was done in order to have room for a 'heigh map' to be overlayed onto a surface. A hight map is like a texture map, however instead of containing colour information, it contains height information. Vertices for each micropolygon are moved to reflect the height point referenced to the corresponding pixel on the height map. This often stretches the micropolygon (And also has the potential to break the $\frac{1}{2}$ pixel size rule). Thus, a micropolygon that would otherwise be offscreen has the potential to be onscreen. While not implemented and beyond the

scope of the paper, height maps can increase the complexity and quality of the result. Room for such a quick implementation improves the outlook for this approach.

Not all objects can be immediately diced. Some objects would produce too many micropolygons, and others start off in a form initially unsuitable for such an operation. For these cases, the object is then split into smaller parts, or 'sliced'. For the case of bicubic patches, this means subdividing them into smaller patches. Given the nature of cubic curves in that they do not follow uniform tesselation, this will most likely produce more suitably spaced micropolygons. As explained in other sections, repeated subdivision operations are too computationally slow for this to be employed often. I would expect, though, that a normal scene would require few objects to be sliced.

Once all objects are diced, the original world and object coordinates need not be considered any longer (unless the previous suggestion of keeping micorpolygon data between frames is considered). Thus there is no need for any inverse operations. Since we are now dealing with screen space, micropolygons that fall entirely outside the viewing field can be culled. It is at this point, also, that micropolygons falling outside the hither-yon range (i.e. they are either too close or too far away from the 'eye') are culled. While not mentioned, backface culling should be able to be performed at this point as well.

A depth buffer is used to distinguish micropolygons which fall under the same pixel. This need be the only form of collission detection used, given the size of micropolygons. It is suggested that lighting and texture calculations be performed prior to depth calculations, however I stipulate that this need not be the case. As mentioned before, the intention was to cut down on memory thrashing and disk access. This need not be the case, as sufficient memory is now available to store everything necessary in RAM. Another reason was to allow extensibility and compatability with other engines. It was envisioned that some objects be rendered using other methods, such as ray-tracing, due to requirements such as reflection. This is no longer necessary.

Shading is all that is left to produce the image. It is suggested that certain filtering be performed to reduce ailising, however this will decrease performance. Calculation of light intensity is easily done using the same method as used in traditional 3D engines. Although mapping such as texturing and height mapping will have already been

performed, extra mapping such as glossiness etc. can also be added at this stage.

This method lends itself to implementation in modern graphics hardware more than the others. This is due to its resemblance to traditional 3D engines as well as features of current standards to accomodate normals (the direction the polygon is facing) in points. A major drawback, though, is the large quantity of micropolygons produced in a normal scene. Even with the suggested changes, the number of points taken into account in each frame are enormous.

Chapter 5

Design

5.1 Comparison of Alternatives

The four alternatives discussed each have different advantages and disadvantages. All of them have a rather long history. At the time of their inception, computer hardware was severely limited. Memory access was synonymous with disk access, which is very different to today, where all mentioned memory operations are done in RAM. Each of these alternatives were intended for non-interactive rendering on reasonably large and expensive systems. The intention here is to compare how they would perform on a normal PC computer system of today's standards.

The scanline method looks good, however several things slow it down considerably. When the curve algorithm fails to converge, a lot of time is spent obtaining an answer. Also, The pre-processing algorithms are fairly lengthy. Particularly, the silhouette detector is a very costly operation, and needs to be performed for every patch, sometimes multiple times.

Forward differencing solves most of the problems associated with the scanline method, however there is a lot of redundancy from the large amount of pixels being re-written so that no gaps appear. Further optimisations can be made, however they do not solve the problem entirely. Especially when translating between curves and surfaces, redundant

pixels are still a huge slowdown.

The cubic frame biffer is quite good in the speed aspect. Each operation is quite simple, and there is no slowdown. Problems can occur, however, when objects are enlarged too much. The main issues is memory. A huge amount of memory is required to store all objects. This has a significant impact on how this method is treated.

Micropolygons suffer for a mixture of the above. Once objects are diced, a large number of micropolygons are produced. This is not enough to run out of memeory, however it could be more than can be processed in a reasonable amount of time. Along with that, significant pre-processing has to be performed before any micropolygons are made. Even with the suggested changes, it still leaves a lot of data to process for each frame.

5.2 Selection

Ironically, the order the alternatives are presented in represents the initial expected performance at the onset of the research. This proved to be inconsistent with the final theoretical analysis. Note that the scope of the research is not to implement all avenues, but rather to choose the best from available information and then compare an implementation to the traditional form.

The objective is to choose an engine that is the fastest on current standard home computers. This is the environment most 3D engines are targeted for. This system has roughly a 3 GHz CPU, 200 GB HDD, 1 GB RAM and a graphics card that has 256 MB RAM. Ideally it should run smooth on such a system. The system obtained for the test has slightly lower specifications, but enough that a real-time test is feaseable.

The cubic frame buffer engine simply takes too much memory. Models would have to be stored on disk, making access times horrendous. Even though it's fast, it's not practical at this point in time. Micropolygons are also too cumbersome. Objects are split into thousands of micropolygons, making both the splitting task and the remaining tasks extremely slow.

Both the scanline and the forward differencing methods have great appeal. Adopting a scanline guarantees no missing pixels and no redundant pixels. Forward differencing guarantees not to default to significantly slower and more painful means and does not require pre-processing to find silhouettes etc.

The speedup produced by the scanline order is now fairly irrelevent, as stated before. Not only that, but the scanline order will prove more computationally expensive to implement with textures. Along with the additional ways of speeding up its implementation, it can be easily seen to be faster than the scanline method. Detailed algorithms are available which add extra speed boosts on top of the additional methods which utilise bit shifting and clever use of registers.

5.3 Integration with Adaptive Subdivision

The main difficulty with a surface representation using forward differencing is that the inaccuracies accumulate too much. The only solution by (Klassen 1994) and all the others the address this issue is to increase the register width. At the time, this involved joining two 32-bit registers together to form a pseudo-64-bit register. This operation takes a lot of time, and thus reduces the efficiency of the algorithm too much.

According to (Klassen 1994), adaptive subdivision is the appropriate alternative. Like forward differencing, points with a fixed step size are chosen. Unlike forward differencing, these points are then used to split the curve. Thus instead of one curve, an arbitary number of curves is produced. Since the inaccuracies are related to the number of steps, and the number of steps is related to the curve size, the number of curves produces can be talored to the size of the curve.

Another advantage of this method is that it helps monotonise the resultant spacing of points found. Normally curves have sections which have a higher density of points. Using forward differencing, the step size can be reduced to accomodate these sections.

Subdivision is reasonably slow, however. New curves are created, and more slower and traditional methods have to be utilised to split the curves in the first place. This

slowdown, however, is still less than that produced by having to joing two registers together. With the advent of common 64-bit registers, the slowdown of joining two 32-bit registers becomes non-existent. Everything can be done in a single register. Thus, this implementation becomes reasonably slow due to the disappearance of the tradeoff.

5.4 Adaptive Forward Differencing

Adaptive forward differencing is mentioned in quite a few papers. It is put best, however, in (Lien, Shantz & Pratt 1987). Several transformations have been developed to halve the step size 5.1 or double it 5.2. This goes back to first principles of differentiation. If you had initially decreased or increased the step size to start off with, it would work this way. Thus, to re-arrange the calculations, simple algebra can be employed.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix} \tag{5.1}$$

$$L^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{8} & \frac{1}{16} \\ 0 & 0 & \frac{1}{4} & -\frac{1}{8} \\ 0 & 0 & 0 & \frac{1}{8} \end{bmatrix} \tag{5.2}$$

The change in step size is triggered by the distance the current pixels is from the previous one. If a pixel is missed, a half-step (or step-down) is performed on the previous result to obtain the next correct pixel. If the step size results in a new pixel that is less than half a pixel away from the previous one, a double-step (or step-up) is performed.

In this way, far fewer redundant pixels are produced while still covering the entire curve. An extra check has to be performed after each pixel is calculated, however after testing it has been found that a step-up or step-down is rare. Other than this, the algorithm

works with the precision and speed of ordinary forward differencing.

Adaptive forward differencing is thus the preferred choice for implementation, and will be considered for the rest of the paper.

5.5 Other Factors and Optimisations

It is possible to combine the two optimisations, however there would be no discernable speed increase (and would most likely produce a speed decrease), as both overheads are added and the advantages overlap.

(Elber & Cohen 1996) provides an interesting optimisation. As it is common for free-form surfaces to be concave, often adjacent curves overlap to produce a very large amount of redundant pixels. Since we cannot skip curves (as that would result in a gap at the ends), that is usually just assumed to be a result of the technique used. (Elber & Cohen 1996) has used a technique to stop curves prematurely. In this way, when a concave surface's curve starts to overlap the previous one, it is terminated at that end. While not mentioned in the paper, it would be presumable that the curve then needs to be started from the other end to avoid a possible gap there. This method actually saves as many pixels as adaptive forward differencing, and is thus worthy of consideration.

Chapter 6

Implementation

6.1 Hardware and Software Implementation

(Chang, Shantz & Rocchetti 1989) provides a very neat template for implementing adaptive forward differencing. 32-bit registers are laid out and are complete for a curve implementation. While 64-bit registers will be needed for surfaces, the method can conceivably be easily transfered from curves to surfaces. This will take an extra dimension, and thus an extra matrix operation. Since the research mainly deals with the 32-bit section, and due to hardware supply problems, the 32-bit registers and associated curves will be discussed.

Matrices can be converted into computations by cross-multiplying. With this, a few optimisations can be performed due to ordering. Register shifts can also be used to squeeze precision out, resulting in a fixed-point format. With the speed of processors, this allows for equivalent efficiency over iteger precision.

6.1.1 Hardware Selection

As hinted before, only a 64-bit processor will be viable for such an implementation. (Chang et al. 1989) provides information on a 64-bit implementation, which would be ideal. Unfortunately this was not possible in our test due to a hardware supply problem.

The techniques can still be used in a higher-level language, however the efficiency will suffer.

With 64-bit registers, greater freedom in terms of precision is possible. While it is mostly fixed for surface calculations, curve calculations have a wide birth of freedom. Unfortunately when testing, the implementation did not live up to this expectation. Tests conclude that the accuracy is too low to be useable. Further tests were thus not possible. Speed tests on curves could still be performed, however.

6.1.2 Software Selection

Due to the lack of timely resources, it was decided to implement the algorithm in C/C++. Classes are not necessary, however certain advantages from C++ notation were appreciated. The implementation is included in Appendix B.

An API called SDL was used for input/output. SDL is an open-source set of libraries that also include OpenGL. The advantage of using SDL is that it has compatability for both the Linux and Windows operating systems as well as support for 2D and 3D acceleration. While 3D acceleration is not an issue given this situation, 2D acceleration is appreciated. Unfortunatly most of that acceleration was lots in the implementation, and resulted in custom memory access functions. It can still be considered a fast implementation given the resources.

Unfortunately there was not enough time to finish the software component. The framework is completely done, and the bezier points (and associations between them) can be seen. Normal 3D operations can be performed on the view, making it a 3D engine in its own right.

6.2 Overflow Prevention

Due to the use of 32-bit registers and the high demands for screen size and view space, it is possible to have a register overflow. For this reason, our fixed-point algorithm

shifts the registers to prevent this. Unlike some other implementations, our algorithm allocates more bits for the significand. This is reflected in the register shifts.

Early tests resulted in a lack of precision that indicated an overflow. This was before the register shifts were introduced. Thus, register shifts are an essential component in obtaining a workable solution.

Chapter 7

Conclusion

Unfortunately the alternative 3D engine method did not live up to expectations. The 32-bit register implementation lacked the needed precision given the methods used. It is possible further register shifting can solve this problem, however not enough time was allowed to test this. Vital computer parts were significantly delayed enough to disallow adequate time for implementation and testing. The author also sufferd a prolonged sickness during the implementation phase that delayed the project significantly.

Partial results, as explained before, were obtained. Apart from the lack of precision, the performance was adequate. The object was viewable in real-time with a forward difference algorithm with a sufficient density to disallow pixel gaps. That being said, a surface algorithm was not developed. While not enough progress was performed to confirm this, it can be reasonably argued that all thing considered, this implementation would not stand up to the traditional polygonal method. Even given further optimisations, such a high accuracy of image at a decent frame-rate is unachieveable.

It could also be considered, though, that a more inaccurate representation using similar techniques to the ones presented in this paper could be a viable candidate to be called a next-generation 3D graphics engine. This would be an area of further research.

References

Abram, G. & Westover, L. (1985), Efficient alias-free rendering using bit-masks and look-up tables, *in* 'Proceedings of the 27th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, San Francisco California, pp. 53–59.

Amburn, P., Grant, E. & Whitted, T. (1986), 'Managing geometric complexity with enhanced procedural models', *Computer Graphics* **20**(4), 189–195.

AMD (2002), *AMD x86-64 Architecture Programmers Manual*, Vol. 1, 3.07 edn, Advanced Micro Devices Inc., Sunnyvale California.

Bajaj, C., Chen, J. & Xu, G. (1995), 'Modeling with cubic a-patches', *ACM Transactions on Graphics* **14**(2), 103–133.

Bishop, G. & Weimer, D. (1986), 'Fast phong shading', *Computer Graphics* **20**(4), 103–106.

Blinn, J. (1978), A scan line algorithm for displaying parametrically defined surfaces, *in* 'Proceedings of the 5th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Atlanta Georgia, p. 27.

Blinn, J. (1982), 'A generalization of algebraic surface drawing', *ACM Transactions on Graphics* **1**(3), 235–256.

Carpenter, L. (1984), 'The a-buffer, an antialiased hidden surface method', *Computer Graphics* **18**(3), 103–108.

Catmull, E. & Smith, A. (1980), 3-d transformations of images in scanline order, *in* 'Proceedings of the 7th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Seattle Washington, pp. 279–285.

Chang, S., Shantz, M. & Rocchetti, R. (1989), 'Rendering cubic curves and surfaces with integer adaptive forward differencing', *Computer Graphics* **23**(3), 157–166.

Cheng, F. & Lin, C. (1985), Clipping of bzier curves, *in* 'Proceedings of the 1985 ACM annual conference on The range of computing: mid-80s perspective', Association for Computing Machinery, Denver Colorado, pp. 74–84.

Chhugani, J. & Kumar, S. (2003), Budget sampling of parametric surface patches, *in* 'Proceedings of the 2003 symposium on Interactive 3D graphics', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Monterey California, pp. 131–138, 244.

Chiyokura, H. & Kimura, F. (1983), 'Design of solids with free-form surfaces', *Computer Graphics* **17**(3), 289–298.

Christensen, P., Stollnitz, E., Salesin, D. & DeRose, T. (1990), 'Global illumination of glossy environments using wavelets and importance', *ACM Transactions on Graphics* **15**(1), 37–71.

Cook, R., Carpenter, L. & Catmull, E. (1987), 'The reyes image rendering architecture', *Computer Graphics* **21**(4), 95–102.

DeRose, T. & Barsky, B. (1988), 'Geometric continuity, shape parameters, and geometric constructions for catmull-rom splines', *ACM Transactions on Graphics* **7**(1), 1–41.

Duff, T. (1992), 'Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry', *Computer Graphics* **26**(2), 131–138.

Earnshaw, R. (1977), Line generation for incremental and raster devices, *in* 'Proceedings of the 4th annual conference on Computer graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, San Jose California, pp. 199–205.

Efremov, A., Havran, V. & Seidel, H. (2005), Robust and numerically stable bzier clipping method for ray tracing nurbs surfaces, *in* 'Proceedings of the 21st spring conference on Computer graphics', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Budmerice, pp. 127–135.

Elber, G. & Cohen, E. (1996), 'Adaptive isocurve-based rendering for freeform surfaces', *ACM Transactions on Graphics* **15**(3), 249–263.

Forsey, D. & Bartels, R. (1988), 'Hierarchical b-spline refinement', *Computer Graphics* **22**(4), 205–212.

Fournier, A. & Fussell, D. (1980), 'Stochastic modeling in computer graphics', *Preliminary papers to be published in Communications of the ACM* **14**(SI), 1–8.

Frisken, S., Perry, R., Rockwood, A. & Jones, T. (2000), Adaptively sampled distance fields: A general representation of shape for computer graphics, *in* 'Proceedings of the 27th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, New Orleans California, pp. 249–254.

Goldman, R. (2002), 'On the algebraic and geometric foundations of computer graphics', *ACM Transactions on Graphics* **21**(1), 52–86.

Grocker, G. (1984), 'Invisibility coherence for fast scan-line hidden surface algorithms', *Computer Graphics* **18**(3), 95–102.

Han, S. & Medioni, G. (1996), Triangular nurbs surface modeling of scattered data, *in* 'Proceedings of the 7th conference on visualization 96', ACM Special Interest Group on Computer Graphics and Interactive Techniques, San Francisco California, pp. 295–302.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. & Stuetzle, W. (1992), 'Surface reconstruction from unorganised points', *Computer Graphics* **26**(2), 71–78.

Kaufman, A. (1987), 'Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes', *Computer Graphics* **21**(4), 171–179.

Kaufman, A. & Shimony, E. (1986), 3d scan-conversion algorithms for voxel-based graphics, *in* 'Proceedings of the 1986 workshop on Interactive 3D graphics', ACM

Special Interest Group on Computer Graphics and Interactive Techniques, Chapel Hill North Carolina, pp. 45–75.

Klassen, V. (1991*a*), 'Drawing antialiased cubic spline curves', *ACM Transactions on Graphics* **10**(1), 92–108.

Klassen, V. (1991*b*), 'Integer forward differencing of cubic polynomials- analysis and algorithms', *ACM Transactions on Graphics* **10**(2), 152–181.

Klassen, V. (1994), 'Exact integer hybrid subdivision and forward differencing of cubics', *ACM Transactions on Graphics* **13**(3), 240–255.

Kreeger, K. & Kaufman, A. (1999), Hybrid volume and polygon rendering with cube hardware, *in* 'SIGGRAPH/Eurographics Workshop on Graphics Hardware', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Los Angeles California, pp. 15–24, 138.

Lane, J., Carpenter, L., Whitted, T. & Blinn, J. (1980), 'Scan line methods for displaying parametrically defined surfaces', *Communications of the ACM* **23**(1), 23–34.

Levoy, M. & Whitted, T. (1985), The use of points as a display primitive, Technical Report 85-022, University of North Carolina, Chapel Hill.

Li, F. & Lau, R. (1999), Real-time rendering of deformable parametric free-form surfaces, *in* 'Proceedings of the ACM symposium on Virtual reality software and technology', ACM Special Interest Group on Computer Graphics and Interactive Techniques and ACM Special Interest Group on Computer-Human Interaction, London, pp. 131–138.

Lien, S., Shantz, M. & Pratt, V. (1987), 'Adaptive forward differencing for rendering curves and surfaces', *Computer Graphics* **21**(4), 111–118.

Loop, C. & DeRose, T. (1989), 'A multisided generalization of bzier surfaces', *ACM Transactions on Graphics* **8**(3), 204–234.

Menon, J. (1993), An introduction to constructive shell representations for free-form surfaces and solids, *in* 'Proceedings of the second ACM symposium on Solid modeling and applications', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Montreal Quebec, pp. 23–34.

Neuerburg, K. (2003), Bzier curves, *in* 'Proceedings of the Spring 2003 Louisiana-Mississippi Section of the Mathematical Association of America', Louisiana-Mississippi Section of the Mathematical Association of America, Clinton Mississippi.

Ramamoorthi, R. & Barr, A. (1997), Fast construction of accurate quaternion splines, *in* 'Proceedings of the 24th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Los Angeles California, pp. 287–292.

Reinhard, E., Shirley, P. & Hansen, C. (2001), Parallel point projection, *in* 'Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics', Institute of Electrical and Electronic Engineers, San Diego California, pp. 29–35.

Schweitzer, D. & Cobb, E. (1982), 'Scanline rendering of parametric surfaces', *Computer Graphics* **16**(3), 265–271.

Sederberg, T. (1995), 'Point and tangent computation of tensor product rational bzier surfaces', *Computer-Aided Geometric Design* **12**(1), 103–106.

Sederberg, T. & Zundel, A. (1989), 'Scan line display of algebraic surfaces', *Computer Graphics* **23**(3), 147–156.

Shantz, M. & Chang, S. (1988), 'Rendering trimmed nurbs with adaptive forward differencing', *Computer Graphics* **22**(4), 189–198.

Vashnav, H. & Rockwood, A. (1993), Calculating offsets of a bezier curve, *in* 'Proceedings of the second ACM symposium on Solid modeling and applications', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Montreal Quebec, pp. 491–492.

Vlasic, D. (2002), Fake phong shading, Masters thesis, The Massachusetts Institute of Technology, Massachusetts.

Von Herzen, B., Barr, A. & Zatz, H. (1990), 'Geometric collisions for time-dependent parametric surfaces', *Computer Graphics* **24**(4), 39–48.

Walia, E. & Singh, C. (2003), Bi-quadratic interpolation of intensity for fast shading of three dimensional objects, *in* 'Proceedings of the Image and Vision Computing

2003 New Zealand', Image and Vision Computing New Zealand, Palmerston North, pp. 96–101.

Warren, J. (1992), 'Creating multisided rational bzier surfaces using base points', *ACM Transactions on Graphics* **11**(2), 127–139.

Watson, B. & Hodges, L. (1992), Algorithms for rendering cubic curves, Technical report, Georgia Institute of Technology, Georgia.

Whitted, T. (1978), A scan line algorithm for computer display of curved surfaces, *in* 'Proceedings of the 5th annual conference on Computer Graphics and interactive techniques', ACM Special Interest Group on Computer Graphics and Interactive Techniques, Atlanta Georgia, p. 26.

Wu, S., Abel, J. & Greenberg, D. (1977), 'An interactive computer graphics approach to surface representation', *Communications of the ACM* **20**(10), 703–712.

Appendix A

Project Specification

University of Southern Queensland
Faculty of Engineering and Surveying

ENG 4111/4112 Research Project
PROJECT SPECIFICATION

CONFIDENTIAL

FOR:	**Justin John Cameron**
TOPIC:	Next-generation 3D graphics engine design
SUPERVISOR:	Prof. John Leis
PROJECT AIM:	This project aims to investigate alternative approaches to 3D engines that do not utilise polygons, and then implement a new design in a low-level programming language.

PROGRAMME: **Issue A, 27th March 2006**

1. Research existing alternatives to polygonal 3D engines to determine their viability for high-speed applications
2. Analyse the strengths and weaknesses of the most viable alternatives
3. Develop a plan for a new engine that draws from the strengths of these previous engines with the emphasis on speed
4. Research a low-level programming language capable of implementing this engine efficiently.
5. Implement the core routines of the engine
6. Implement the base framework for the engine so it can be utilised by an application

As time permits:

7. Implement a standard interface for the engine so it can be separated from the application
8. Implement a test application to determine its capabilities
9. Analyse the performance of the engine using this application

AGREED: _____(Student) _____(Supervisor)

(dated) ____/____/____. (dated) ____/____/____.

Appendix B

Code Listing

This appendix contains the complete listing of the program developed to implement the chosen 3D graphics engine design. The interface used to access graphics hardware and input devices is SDL (Simple Directmedia Layer). Thus, SDL must be installed on the system. In order to comply with the LGPL license (A fairly standard license for open systems), the library libsdl.so must be available for modification by users.

B.1 Makefile

Listing B.1: Makefile for test program

```
OUTPUT = program

FLAGS = -Wall `sdl-config --cflags` -lSDL
FINALFLAGS = -O3 -ffast-math -funroll-all-loops
LIBS =

debug:
	@g++ main.cpp -o $(OUTPUT) $(FLAGS) $(LIBS)
program:
	@g++ main.cpp -o $(OUTPUT) $(FINALFLAGS) $(FLAGS) $(LIBS)

clean:
	@rm program
```

B.2 Data and Variables

This file, `vars.h`, contains all the customiseable variables. Included also is the data for creating the model that is used, the infamous Utah Teapot.

Listing B.2: The header file containing the model data and the variables

```
#ifndef __ATTRIBUTES_H_
#define __ATTRIBUTES_H_

#include <SDL/SDL.h>


// Changeable Variables
// --------------------

// Screen mode: 1: 800x600, 2: 1024x760
#define J_SCREEN_SIZE 2
// Mouse Sensitivity (Default-10)
#define J_MOUSE_SENSITIVITY 10
// Keyboard Sensitivity (Default-10)
#define J_KEY_SENSITIVITY 10


// End of changeable variables

#if J_SCREEN_SIZE == 2
        #define J_SCREEN_WIDTH 1024
        #define J_SCREEN_HEIGHT 768
#else
        #define J_SCREEN_WIDTH  800
        #define J_SCREEN_HEIGHT 600
#endif

#define J_CENTRE_X (J_SCREEN_WIDTH) / 2
#define J_CENTRE_Y (J_SCREEN_HEIGHT) / 2

// Defines the sample model using bicubic patches (the ORIGINAL Utah teapot)

#define J_NUM_PATCHES 28
#define J_NUM_CURVES 4
#define J_NUM_POINTS 4
#define J_NUM_DIMENTIONS 3
```

```
// Array: Number of patches; 4 curves/patch; 4 points/curve; 3 dimentions/poi
double patcharray[J_NUM_PATCHES][J_NUM_CURVES][J_NUM_POINTS][J_NUM_DIMENTIONS
  {
    {{1.4, 2.25, 0},          {1.3375, 2.38125, 0},          {1.4375, 2.38125, 0}
{1.5, 2.25, 0}},
    {{1.4, 2.25, 0.784},      {1.3375, 2.38125, 0.749},      {1.4375, 2.38125, 0.
{1.5, 2.25, 0.84}},
    {{0.784, 2.25, 1.4},      {0.749, 2.38125, 1.3375},      {0.805, 2.38125, 1.4
{0.84, 2.25, 1.5}},
    {{0, 2.25, 1.4},          {0, 2.38125, 1.3375},          {0, 2.38125, 1.4375}
{0, 2.25, 1.5}}
  }, {
    {{0, 2.25, 1.4},          {0, 2.38125, 1.3375},          {0, 2.38125, 1.4375}
{0, 2.25, 1.5}},
    {{-0.784, 2.25, 1.4},     {-0.749, 2.38125, 1.3375},     {-0.805, 2.38125, 1.
{-0.84, 2.25, 1.5}},
    {{-1.4, 2.25, 0.784},     {-1.3375, 2.38125, 0.749},     {-1.4375, 2.38125, 0
{-1.5, 2.25, 0.84}},
    {{-1.4, 2.25, 0},         {-1.3375, 2.38125, 0},         {-1.4375, 2.38125, 0
{-1.5, 2.25, 0}}
  }, {
    {{-1.4, 2.25, 0},         {-1.3375, 2.38125, 0},         {-1.4375, 2.38125, 0
{-1.5, 2.25, 0}},
    {{-1.4, 2.25, -0.784},    {-1.3375, 2.38125, -0.749},    {-1.4375, 2.38125, -
{-1.5, 2.25, -0.84}},
    {{-0.784, 2.25, -1.4},    {-0.749, 2.38125, -1.3375},    {-0.805, 2.38125, -
{-0.84, 2.25, -1.5}},
    {{0, 2.25, -1.4},         {0, 2.38125, -1.3375},         {0, 2.38125, -1.4375
{0, 2.25, -1.5}}
  }, {
    {{0, 2.25, -1.4},         {0, 2.38125, -1.3375},         {0, 2.38125, -1.4375
{0, 2.25, -1.5}},
    {{0.784, 2.25, -1.4},     {0.749, 2.38125, -1.3375},     {0.805, 2.38125, -1
{0.84, 2.25, -1.5}},
```

```
    {{1.4, 2.25, −0.784},
{1.5, 2.25, −0.84}},
    {{1.4, 2.25, 0},
{1.5, 2.25, 0}}
  }, {
    {{1.5, 2.25, 0},
{2, 0.75, 0}},
    {{1.5, 2.25, 0.84},
{2, 0.75, 1.12}},
    {{0.84, 2.25, 1.5},
{1.12, 0.75, 2}},
    {{0, 2.25, 1.5},
{0, 0.75, 2}}
  }, {
    {{0, 2.25, 1.5},
{0, 0.75, 2}},
    {{−0.84, 2.25, 1.5},
{−1.12, 0.75, 2}},
    {{−1.5, 2.25, 0.84},
{−2, 0.75, 1.12}},
    {{−1.5, 2.25, 0},
{−2, 0.75, 0}}
  }, {
    {{−1.5, 2.25, 0},
{−2, 0.75, 0}},
    {{−1.5, 2.25, −0.84},
{−2, 0.75, −1.12}},
    {{−0.84, 2.25, −1.5},
{−1.12, 0.75, −2}},
    {{0, 2.25, −1.5},
{0, 0.75, −2}}
  }, {
    {{0, 2.25, −1.5},
{0, 0.75, −2}},
```

```
{1.3375, 2.38125, −0.749},
{1.3375, 2.38125, 0},
{1.75, 1.725, 0},
{1.75,1.725, 0.98},
{0.98, 1.725, 1.75},
{0, 1.725, 1.75},
{0, 1.725, 1.75},
{−0.98, 1.725, 1.75},
{−1.75, 1.725, 0.98},
{−1.75, 1.725, 0},
{−1.75, 1.725, 0},
{−1.75, 1.725, −0.98},
{−0.98, 1.725, −1.75},
{0, 1.725, −1.75},
{0, 1.725, −1.75},
```

```
{1.4375 2.38125, −
{1.4375 2.38125,0},
{2, 1.2 0},
{2, 1.2 1.12},
{1.12, 2, 2},
{0, 1.2 , 2},
{0, 1.2, 2},
{−1.12, 1.2, 2},
{−2, 1.2, 1.12},
{−2, 1.2, 0},
{−2, 1.2, 0},
{−2, 1.2, −1.12},
{−1.12, 1.2, −2},
{0, 1.2, −2},
{0, 1.2, −2},
```

```
    {{0.84, 2.25, -1.5},
{1.12, 0.75, -2}},
    {{1.5, 2.25, -0.84},
{2, 0.75, -1.12}},
    {{1.5, 2.25, 0},
{2, 0.75, 0}}
   },
   {
    {{2, 0.75, 0},
{1.5, 0, 0}},
    {{2, 0.75, 1.12},
{1.5, 0, 0.84}},
    {{1.12, 0.75, 2},
{0.84, 0, 1.5}},
    {{0, 0.75, 2},
{0, 0, 1.5}}
   },
   {
    {{0, 0.75, 2},
{0, 0, 1.5}},
    {{-1.12, 0.75, 2},
{-0.84, 0, 1.5}},
    {{-2, 0.75, 1.12},
{-1.5, 0, 0.84}},
    {{-2, 0.75, 0},
{-1.5, 0, 0}}
   },
   {
    {{-2, 0.75, 0},
{-1.5, 0, 0}},
    {{-2, 0.75, -1.12},
{-1.5, 0, -0.84}},
    {{-1.12, 0.75, -2},
{-0.84, 0, -1.5}},
    {{0, 0.75, -2},
{0, 0, -1.5}}
   }, {
```

```
{0.98, 1.725, -1.75},
{1.75, 1.725, -0.98},
{1.75, 1.725, 0},
{2, 0.3, 0},
{2, 0.3, 1.12},
{1.12, 0.3, 2},
{0, 0.3, 2},
{0, 0.3, 2},
{-1.12, 0.3, 2},
{-2, 0.3, 1.12},
{-2, 0.3, 0},
{-2, 0.3, 0},
{-2, 0.3, -1.12},
{-1.12, 0.3, -2},
{0, 0.3, -2},
```

```
{1.12, 2, -2},
{2, 1.2, -1.12},
{2, 1.2, 0},
{1.5, 0.75, 0},
{1.5, 0.75, 0.84},
{0.84, 0.075, 1.5},
{0, 0.075, 1.5},
{0, 0.075, 1.5},
{-0.84, 0.075, 1.5},
{-1.5, 0.075, 0.84},
{-1.5, 0.075, 0},
{-1.5, 0.075, 0},
{-1.5, 0.075, -0.84},
{-0.84, 0.075, -1.5},
{0, 0.075, -1.5},
```

```
    {{0, 0.75, -2},
{0, 0, -1.5}},
    {{1.12, 0.75, -2},
{0.84, 0, -1.5}},
    {{2, 0.75, -1.12},
{1.5, 0, -0.84}},
    {{2, 0.75, 0},
{1.5, 0, 0}}
   }, {
    {{-1.6, 1.875, 0},
{-2.7, 1.65, 0}},
    {{-1.6, 1.875, 0.3},
{-2.7, 1.65, 0.3}},
    {{-1.5, 2.1, 0.3},
{-3, 1.65, 0.3}},
    {{-1.5, 2.1, 0},
{-3, 1.65, 0}}
   }, {
    {{-1.5, 2.1, 0},
{-3, 1.65, 0}},
    {{-1.5, 2.1, -0.3},
{-3, 1.65, -0.3}},
    {{-1.6, 1.875, -0.3},
{-2.7, 1.65, -0.3}},
    {{-1.6, 1.875, 0},
{-2.7, 1.65, 0}}
   }, {
    {{-2.7, 1.65, 0},
{-2, 0.75, 0}},
    {{-2.7, 1.65, 0.3},
{-2, 0.75, 0.3}},
    {{-3, 1.65, 0.3},
{-1.9, 0.45, 0.3}},
    {{-3, 1.65, 0},
{-1.9, 0.45, 0}}
   }, {
    {{0, 0.3, -2},
{1.12, 0.3, -2}},
    {{2, 0.3, -1.12},
{2, 0.3, 0}},
    {{-2.3, 1.875, 0},
{-2.3, 1.875, 0.3}},
    {{-2.5, 2.1, 0.3},
{-2.5, 2.1, 0}},
    {{-2.5, 2.1, 0},
{-2.5, 2.1, -0.3}},
    {{-2.3, 1.875, -0.3},
{-2.3, 1.875, 0}},
    {{-2.7, 1.425, 0},
{-2.7, 1.425, 0.3}},
    {{-3, 1.2, 0.3},
{-3, 1.2, 0}},
    {{0, 0.075, -1.5},
{0.84, 0.075, -1.5}},
    {{1.5, 0.075, -0.84},
{1.5, 0.075, 0}},
    {{-2.7, 1.875, 0},
{-2.7, 1.875, 0.3}},
    {{-3, 2.1, 0.3},
{-3, 2.1, 0}},
    {{-3, 2.1, 0},
{-3, 2.1, -0.3}},
    {{-2.7, 1.875, -0.3},
{-2.7, 1.875, 0}},
    {{-2.5, 0.975, 0},
{-2.5, 0.975, 0.3}},
    {{-2.65, 0.7875, 0.3},
{-2.65, 0.7875, 0}},
```

```
{−1.9, 0.45, 0}}
  }, {
    {{−3, 1.65, 0},
  {−1.9, 0.45, 0}},
    {{−3, 1.65, −0.3},
  {−1.9, 0.45, −0.3}},
    {{−2.7, 1.65, −0.3},
  {−2, 0.75, −0.3}},
    {{−2.7, 1.65, 0},
  {−2, 0.75, 0}}
  }, {
    {{1.7, 1.275, 0},
  {2.7, 2.25, 0}},
    {{1.7, 1.275, 0.66},
  {2.7, 2.25, 0.25}},
    {{1.7, 0.45, 0.66},
  {3.3, 2.25, 0.25}},
    {{1.7, 0.45, 0},
  {3.3, 2.25, 0}}
  }, {
    {{1.7, 0.45, 0},
  {3.3, 2.25, 0}},
    {{1.7, 0.45, −0.66},
  {3.3, 2.25, −0.25}},
    {{1.7, 1.275, −0.66},
  {2.7, 2.25, −0.25}},
    {{1.7, 1.275, 0},
  {2.7, 2.25, 0}}
  }, {
    {{2.7, 2.25, 0},
  {2.8, 2.25, 0}},
    {{2.7, 2.25, 0.25},
  {2.8, 2.25, 0.15}},
    {{3.3, 2.25, 0.25},
```

```
{−2.65, 0.7875, 0},
  {−3, 1.2, 0},
{−2.65, 0.7875, −0.3},
  {−3, 1.2, −0.3},
{−2.5, 1.425, −0.3},
  {−2.7, 1.425, −0.3},
{−2.5, 1.425, 0},
  {−2.7, 1.425, 0},

{2.3, 1.95, 0},
  {2.6, 1.275, 0},
{2.3, 1.95, 0.25},
  {2.6, 1.275, 0.66},
{2.4, 1.875, 0.25},
  {3.1, 0.675, 0.66},
{2.4, 1.875, 0},
  {3.1, 0.675, 0},

{2.4, 1.875, 0},
  {3.1, 0.675, 0},
{2.4, 1.875, −0.25},
  {3.1, 0.675, −0.66},
{2.3, 1.95, −0.25},
  {2.6, 1.275, −0.66},
{2.3, 1.95, 0},
  {2.6, 1.275, 0},

{2.9, 2.325, 0},
  {2.8, 2.325, 0},
{2.9, 2.325, 0.15},
  {2.8, 2.325, 0.25},
{3.45, 2.3625, 0.15},
  {3.525, 2.34375, 0.25},
```

```
{3.2, 2.25, 0.15}},
    {{3.3, 2.25, 0},
{3.2, 2.25, 0}}
  }, {
    {{3.3, 2.25, 0},
{3.2, 2.25, 0}},
    {{3.3, 2.25, -0.25},
{3.2, 2.25, -0.15}},
    {{2.7, 2.25, -0.25},
{2.8, 2.25, -0.15}},
    {{2.7, 2.25, 0},
{2.8, 2.25, 0}}
  }, {
    {{0, 3, 0},
{0.2, 2.55, 0}},
    {{0, 3, 0.002},
{0.2, 2.55, 0.112}},
    {{0.002, 3, 0},
{0.112, 2.55, 0.2}},
    {{0, 3, 0},
{0, 2.55, 0.2}}
  }, {
    {{0, 3, 0},
{0, 2.55, 0.2}},
    {{-0.002, 3, 0},
{-0.112, 2.55, 0.2}},
    {{0, 3, 0.002},
{-0.2, 2.55, 0.112}},
    {{0, 3, 0},
{-0.2, 2.55, 0}}
  }, {
    {{0, 3, 0},
{-0.2, 2.55, 0}},
    {{0, 3, -0.002},
{3.525, 2.34375, 0},
{3.2, 2.25, 0}}
  }, {
    {{3.3, 2.25, 0},
{3.525, 2.34375, 0},
{3.2, 2.25, 0}},
    {{3.3, 2.25, -0.25},
{3.525, 2.34375, -0.25},
{3.2, 2.25, -0.15}},
    {{2.7, 2.25, -0.25},
{2.8, 2.325, -0.25},
{2.8, 2.325, 0}},
    {{2.7, 2.25, 0},
{2.8, 2.325, 0}}
  }, {
    {{0, 3, 0},
{0.8, 3, 0}},
    {{0, 3, 0.45},
{0.8, 3, 0.45}},
    {{0.002, 3, 0},
{0.45, 3, 0.8}},
    {{0, 3, 0.2},
{0, 3, 0.8}}
  }, {
    {{0, 3, 0},
{0, 3, 0.8}},
    {{-0.002, 3, 0},
{-0.45, 3, 0.8}},
    {{0, 3, 0.002},
{-0.8, 3, 0.45}},
    {{0, 3, 0},
{-0.8, 3, 0}}
  }, {
    {{0, 3, 0},
{-0.8, 3, 0}},
    {{0, 3, -0.002},
{3.45, 2.3625, 0},
{3.2, 2.25, 0}}
  }, {
    {{3.3, 2.25, 0},
{3.45, 2.3625, 0},
{3.2, 2.25, 0}},
    {{3.3, 2.25, -0.25},
{3.45, 2.3625, -0.15},
{3.2, 2.25, -0.15}},
    {{2.7, 2.25, -0.25},
{2.9, 2.325, -0.15},
{2.8, 2.325, 0}},
    {{2.7, 2.25, 0},
{2.8, 2.325, 0}}
  }, {
    {{0, 3, 0},
{0.8, 3, 0}},
    {{0, 3, 0.45},
{0.8, 3, 0.45}},
    {{0.002, 3, 0},
{0.45, 3, 0.8}},
    {{0, 3, 0.2},
{0, 3, 0.8}}
  }, {
    {{0, 3, 0},
{0, 3, 0.8}},
    {{-0.002, 3, 0},
{-0.45, 3, 0.8}},
    {{0, 3, 0.002},
{-0.8, 3, 0.45}},
    {{0, 3, 0},
{-0.8, 3, 0}}
  }, {
    {{0, 3, 0},
{-0.8, 3, 0}},
    {{0, 3, -0.002},
{-0.8, 3, -0.45}},
```

```
{{−0.2, 2.55, −0.112}},
    {{−0.002, 3, 0},
{−0.112, 2.55, −0.2}},
    {{0, 3, 0},
{0, 2.55, −0.2}}
  }, {
    {{0, 3, 0},
{0, 2.55, −0.2}},
    {{0.002, 3, 0},
{0.112, 2.55, −0.2}},
    {{0, 3, −0.002},
{0.2, 2.55, −0.112}},
    {{0, 3, 0},
{0.2, 2.55, 0}}
  }, {
    {{0.2, 2.55, 0},
{1.3, 2.25, 0}},
    {{0.2, 2.55, 0.112},
{1.3, 2.25, 0.728}},
    {{0.112, 2.55, 0.2},
{0.728, 2.25, 1.3}},
    {{0, 2.55, 0.2},
{0, 2.25, 1.3}}
  }, {
    {{0, 2.55, 0.2},
{0, 2.25, 1.3}},
    {{−0.112, 2.55, 0.2},
{−0.728, 2.25, 1.3}},
    {{−0.2, 2.55, 0.112},
{−1.3, 2.25, 0.728}},
    {{−0.2, 2.55, 0},
{−1.3, 2.25, 0}}
  }, {
    {{−0.2, 2.55, 0},
{−0.45, 3, −0.8}},
    {{0, 3, −0.8},
{0, 2.55, −0.2}}
  }, {
    {{0, 3, −0.8},
{0.45, 3, −0.8}},
    {{0.8, 3, −0.45},
{0.8, 3, 0}},
    {{0.4, 2.4, 0},
{0.4, 2.4, 0.224}},
    {{0.224, 2.4, 0.4},
{0, 2.4, 0.4}},
    {{0, 2.4, 0.4},
{0, 2.4, 1.3}}
  }, {
    {{−0.224, 2.4, 0.4},
{−0.4, 2.4, 0.224}},
    {{−0.4, 2.4, 0},
{−0.4, 2.4, 0}}
  }, {
    {{0, 2.7, 0},
{0, 2.7, 0}},
    {{0, 2.7, 0},
{0, 2.7, 0}},
    {{0, 2.7, 0},
{0, 2.7, 0}},
    {{0, 2.7, 0},
{0, 2.7, 0}},
    {{1.3, 2.4, 0},
{1.3, 2.4, 0.728}},
    {{0.728, 2.4, 1.3},
{0, 2.4, 1.3}},
    {{0, 2.4, 1.3},
{−0.728, 2.4, 1.3}},
    {{−1.3, 2.4, 0.728},
{−1.3, 2.4, 0}},
    {{−1.3, 2.4, 0},
{−1.3, 2.4, 0},
```

```
{-1.3, 2.25, 0}},
    {{-0.2, 2.55, -0.112}, {-0.4, 2.4, -0.224}, {-1.3, 2.4, -0.728},
{-1.3, 2.25, -0.728}},
    {{-0.112, 2.55, -0.2}, {-0.224, 2.4, -0.4}, {-0.728, 2.4, -1.3},
{-0.728, 2.25, -1.3}},
    {{0, 2.55, -0.2}, {0, 2.4, -0.4}, {0, 2.4, -1.3},
{0, 2.25, -1.3}}
  }, {
    {{0, 2.55, -0.2}, {0, 2.4, -0.4}, {0, 2.4, -1.3},
{0, 2.25, -1.3}},
    {{0.112, 2.55, -0.2}, {0.224, 2.4, -0.4}, {0.728, 2.4, -1.3},
{0.728, 2.25, -1.3}},
    {{0.2, 2.55, -0.112}, {0.4, 2.4, -0.224}, {1.3, 2.4, -0.728},
{1.3, 2.25, -0.728}},
    {{0.2, 2.55, 0}, {0.4, 2.4, 0}, {1.3, 2.4, 0},
{1.3, 2.25, 0}}
  }
};
#endif
```

B.3 Program Code

This file, `main.cpp`, contains the program. Of course it's incomplete due to time constraints, but it contains all the framework for the intended implementation.

Listing B.3: The C++ program code

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <SDL/SDL.h>
#include "vars.h"

using namespace std;

int depth_buffer_size;
SDL_Surface *screen;
Uint8 depthbuffer[J_SCREEN_WIDTH][J_SCREEN_HEIGHT];
int controlpoints[J_NUM_PATCHES][J_NUM_CURVES][J_NUM_POINTS][J_NUM_DIMENTIONS];
double xferpoints[J_NUM_PATCHES * J_NUM_CURVES * J_NUM_POINTS * J_NUM_DIMENTIONS];
double rotatehoz, rotatevert;
int panhoz, panvert;
double zoom;
bool panninghoz, panningvert, zooming;
bool moving;


bool initialise();
void deinitialise();
void setpixel(int, int, Uint8, Uint8, Uint8, Uint8);
void setpixel(int, int, Uint8, float, float, float, float);
void drawline(int, int, int, int, float, float, float, float);
bool pollaction();
void getscreen();
void freescreen();
void drawstuff();
void setcourse();
void resetcourse(bool, bool, bool, bool, bool);
void changecourse(int, int, int, int, int);
void changedirection();
void engage();
void drawcurve(int[], int[], int[], int[]);
```

```
// Prepares things like the video card, the frame and depth buffers, OpenGL,
bool initialise() {
        const Uint32 video_mode_flags = SDL_HWSURFACE | SDL_OPENGLBLIT | SDL_I

        SDL_WarpMouse(J_CENTRE_X, J_CENTRE_Y);

        if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER | SDL_INIT_EVENTTHREAD) =
                cout<<"SDL Video and Timer Error: "<<SDL_GetError()<<endl;
                return false;
        }

        SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 8);
        SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
        SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
        SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 8);
        SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

        // This is in case the video card doesn't support the right display m
        for(depth_buffer_size = 32; depth_buffer_size >= 8; depth_buffer_size
                SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, depth_buffer_size);
                screen = SDL_SetVideoMode(J_SCREEN_WIDTH, J_SCREEN_HEIGHT, 32
                if(screen != NULL) {
                        break;
                }
        }
        if(depth_buffer_size < 8) {
                cout<<"SDL Video Mode Error: "<<SDL_GetError()<<endl;
                return false;
        }

        /*if(SDL_SetAlpha(screen, SDL_SRCALPHA | SDL_RLEACCEL, SDL_ALPHA_OPAQ
                cout<<"SDL Alpha Warning: "<<SDL_GetError()<<endl;
        }*/

        SDL_ShowCursor(false);
```

```
        setcourse();

        return true;
}

// Is (automatically) called when the program quits
void deinitialise() {

        SDL_ShowCursor(true);

        SDL_Quit();
}

// Fast function for blitting a pixel to the frame buffer using integers but
void setpixel(int x, int y, Uint8 depth, Uint8 red, Uint8 green, Uint8 blue)
        if(x < 0 || y < 0 || x >= J_SCREEN_WIDTH || y >= J_SCREEN_HEIGHT) {
                return;
        }

        Uint8 *bufp = (Uint8 *)screen->pixels + y*screen->pitch + (x << 2);

        if(depthbuffer[x][y] <= depth) {
                return;
        }

        float scale = (float)(255 - depth) / 255;

        bufp[0] = (Uint8)((float)red * scale);
        bufp[1] = (Uint8)((float)green * scale);
        bufp[2] = (Uint8)((float)blue * scale);
        depthbuffer[x][y] = depth;
}

// Slower function for blitting a pixel to the frame buffer using floats (0-1
void setpixel(int x, int y, Uint8 depth, float red, float green, float blue,
        if(x < 0 || y < 0 || x >= J_SCREEN_WIDTH || y >= J_SCREEN_HEIGHT) {
                return;
        }
```

```
    Uint8 *bufp = (Uint8 *)screen->pixels + y*screen->pitch + (x << 2);

    if(depthbuffer[x][y] <= depth) {
        return;
    }

    float oldportion = 1 - alpha;
    float scale = (float)(255 - depth) / 255;

    bufp[0] = (Uint8)(((float)bufp[0] / 255 * oldportion + red * alpha * 
    bufp[1] = (Uint8)(((float)bufp[1] / 255 * oldportion + green * alpha 
    bufp[2] = (Uint8)(((float)bufp[2] / 255 * oldportion + blue * alpha *
    depthbuffer[x][y] = depth;
}

// Bresenham line-drawing algorithm
// adapted from http://www.cit.gu.edu.au/~anthony/info/graphics/bresenham.pro
void drawline(int x1, int y1, int z1, int x2, int y2, int z2, float red, floa
    int dx, dy, dz, l, m, n, x_inc, y_inc, z_inc,
        err_1, err_2, dx2, dy2, dz2,
        pixelx, pixely, pixelz;

    pixelx = x1;
    pixely = y1;
    pixelz = z1;
    dx = x2 - x1;
    dy = y2 - y1;
    dz = z2 - z1;
    x_inc = (dx < 0) ? -1 : 1;
    l = abs(dx);
    y_inc = (dy < 0) ? -1 : 1;
    m = abs(dy);
    z_inc = (dz < 0) ? -1 : 1;
    n = abs(dz);
    dx2 = l << 1;
    dy2 = m << 1;
```

```
dz2 = n << 1;
if((l >= m) && (l >= n)) {
        err_1 = dy2 - l;
        err_2 = dz2 - l;
        for(int i = 0; i < l; i++) {
                setpixel(pixelx, pixely, (Uint8)pixelz, red, green, b
                if(err_1 > 0) {
                        pixely += y_inc;
                        err_1 -= dx2;
                }
                if(err_2 > 0) {
                        pixelz += z_inc;
                        err_2 -= dx2;
                }
                err_1 += dy2;
                err_2 += dz2;
                pixelx += x_inc;
        }
} else if((m >= l) && (m >= n)) {
        err_1 = dx2 - m;
        err_2 = dz2 - m;
        for(int i = 0; i < m; i++) {
                setpixel(pixelx, pixely, (Uint8)pixelz, red, green, b
                if(err_1 > 0) {
                        pixelx += x_inc;
                        err_1 -= dy2;
                }
                if(err_2 > 0) {
                        pixelz += z_inc;
                        err_2 -= dy2;
                }
                err_1 += dx2;
                err_2 += dz2;
                pixely += y_inc;
        }
```

```
    } else {
        err_1 = dy2 - n;
        err_2 = dx2 - n;
        for(int i = 0; i < n; i++) {
            setpixel(pixelx, pixely, (Uint8)pixelz, red, green, b
            if(err_1 > 0) {
                pixely += y_inc;
                err_1 -= dz2;
            }
            if(err_2 > 0) {
                pixelx += x_inc;
                err_2 -= dz2;
            }
            err_1 += dy2;
            err_2 += dx2;
            pixelz += z_inc;
        }
    }
    setpixel(pixelx, pixely, (Uint8)pixelz, red, green, blue, alpha);
}

// SDL event handler
bool pollaction() {
    SDL_Event event;
    static bool ignore = false;

    while(SDL_PollEvent(&event)) {
        // Usually changes the view
        switch(event.type) {
            case SDL_QUIT:
                // When quitting
                return false;
            case SDL_KEYDOWN:
                switch(event.key.keysym.sym) {
                    case SDLK_ESCAPE:
                        // For quit
```

```
    return false;
case SDLK_KP0:
        resetcourse(false, false, tru
        break;
case SDLK_KP1:
        changecourse(0, -1, 1, 0);
        break;
case SDLK_KP2:
        changecourse(0, 0, 1, 0);
        break;
case SDLK_KP3:
        changecourse(0, 1, 1, 0);
        break;
case SDLK_KP4:
        changecourse(0, -1, 0, 0);
        break;
case SDLK_KP5:
        resetcourse(false, false, tru
        break;
case SDLK_KP6:
        changecourse(0, 1, 0, 0);
        break;
case SDLK_KP7:
        changecourse(0, -1, -1, 0)
        break;
case SDLK_KP8:
        changecourse(0, 0, -1, 0);
        break;
case SDLK_KP9:
        changecourse(0, 1, -1, 0);
        break;
case SDLK_KP_PERIOD:
        resetcourse(true, true, true,
        break;
case SDLK_KP_MINUS:
        changecourse(0, 0, 0, -1);
        break;
```

```
            case SDLK_KP_PLUS:
                changecourse(0, 0, 0, 1);
                break;
            case SDLK_KP_ENTER:
                changecourse(0, 0, 0, -1);
                break;
            default:
                break;
        }
        break;
    case SDL_KEYUP:
        switch(event.key.keysym.sym) {
            case SDLK_KP1:
            case SDLK_KP2:
            case SDLK_KP3:
            case SDLK_KP4:
            case SDLK_KP6:
            case SDLK_KP7:
            case SDLK_KP8:
            case SDLK_KP9:
            case SDLK_KP_MINUS:
            case SDLK_KP_PLUS:
            case SDLK_KP_ENTER:
                panninghoz = false;
                panningvert = false;
                zooming = false;
                moving = false;
                break;
            default:
                break;
        }
        break;
    case SDL_MOUSEMOTION:
        if(ignore) {
            ignore = false;
            continue;
        }
```

```
                changecourse(event.motion.x - J_CENTRE_X, eve
                ignore = true;
                SDL_WarpMouse(J_CENTRE_X, J_CENTRE_Y);
                break;
            }
        }
        return true;
    }
    // Hog the frame buffer for blitting
    void getscreen() {
        if (!SDL_MUSTLOCK(screen)) {
            if (SDL_LockSurface(screen) == -1) {
                cout<<"SDL Screen Lock Error: "<<SDL_GetError()<<endl
                exit(-1);
            }
        }
    }
    // Release the frame buffer
    void freescreen() {
        if (!SDL_MUSTLOCK(screen)) {
            SDL_UnlockSurface(screen);
        }
    }
    // Calls the routines for updating the screen
    void drawstuff() {
        getscreen();
        SDL_FillRect(screen, NULL, SDL_MapRGBA(screen->format, 0, 0, 0, 255));
        for(int x = 0; x < J_SCREEN_WIDTH; x++) {
            setpixel(x, 0, 0, 28, 28, 28);
            setpixel(x, J_SCREEN_HEIGHT - 1, 0, 28, 28, 28);
```

```
}
for(int y = 0; y < J_SCREEN_HEIGHT; y++) {
        setpixel(0, y, 0, 28, 28, 28);
        setpixel(J_SCREEN_WIDTH - 1, y, 0, 28, 28, 28);
}

setpixel(0, 0, 0, 255, 255, 255);
setpixel(0, J_SCREEN_HEIGHT - 1, 0, 0, 0, 255);
setpixel(J_SCREEN_WIDTH - 1, J_SCREEN_HEIGHT - 1, 0, 255, 0, 0);
setpixel(J_SCREEN_WIDTH - 1, 0, 0, 255, 0);

if(moving) {
        changecourse(0, 0, 0, 0, 0);
}
engage();

freescreen();

SDL_Flip(screen);
SDL_GL_SwapBuffers();    // For double-buffering
}

// Firing up
int main() {

        atexit(deinitialise);
        if(!initialise()) {
                cout<<"Could not initialise program, exiting...."<<endl;
                exit(-1);
        }

        while(pollaction()) {
                drawstuff();
        }

}
```

```
// Initialise the view
void setcourse() {
        // Normalises the values
        for(unsigned int i = 1; i < J_NUM_PATCHES * J_NUM_CURVES * J_NUM_POIN
                ((double *)patcharray)[i] -= 1.5;
        }

        resetcourse(true, true, true, true, true);
}
// Resets aspects of the view
void resetcourse(bool hozrotate, bool vertrotate, bool hozpan, bool vertpan,
        bool rotatingorzooming = false;
        if(hozrotate) {
                rotatehoz = 0;
                rotatingorzooming = true;
        }
        if(vertrotate) {
                rotatevert = 0;
                rotatingorzooming = true;
        }
        if(hozpan) {
                panhoz = 0;
                panninghoz = false;
        }
        if(vertpan) {
                panvert = 0;
                panningvert = false;
        }
        if(zoomamount) {
                zoom = 1;
                zooming = false;
                rotatingorzooming = true;
        }
```

```
        if(panninghoz == false && panningvert == false && zooming == false) {
            moving = false;
        }

        if(rotatingorzooming) {
            changedirection();
        }

        changecourse(0, 0, 0, 0, 0);
    }

    // Update the view for specific values
    void changecourse(int hozrotate, int vertrotate, int hozpan, int vertpan, int
        static int hozpanamount, vertpanamount, zoomingamount;
        bool rotatingorzooming = false;

        if(hozrotate) {
            rotatehoz += (double)hozrotate * ((double)J_MOUSE_SENSITIVITY
            rotatingorzooming = true;
        }
        if(vertrotate) {
            rotatevert += (double)vertrotate * ((double)J_MOUSE_SENSITIVI
            rotatingorzooming = true;
        }
        if(hozpan) {
            hozpanamount = hozpan;
            panninghoz = true;
            moving = true;
        }
        if(panninghoz) {
            panhoz -= hozpanamount * J_KEY_SENSITIVITY;
        }
        if(vertpan) {
            vertpanamount = vertpan;
            panningvert = true;
            moving = true;
        }
```

```
    if(panningvert) {
        panvert -= vertpanamount * J_KEY_SENSITIVITY;
    }
    if(zoomamount) {
        zoomingamount = zoomamount;
        zooming = true;
        moving = true;
    }
    if(zooming) {
        zoom += zoomingamount * ((double)J_KEY_SENSITIVITY / 130);
        rotatingorzooming = true;
    }

    if(rotatingorzooming) {
        changedirection();
    }

    for(unsigned int i = 0; i < J_NUM_PATCHES * J_NUM_CURVES * J_NUM_POIN
            ((Uint32 *)controlpoints)[i] = J_CENTRE_X - (Uint32)(xferpoin
            ((Uint32 *)controlpoints)[i+1] = J_CENTRE_Y - (Uint32)(xferpo
            ((Uint32 *)controlpoints)[i+2] = 128 - (Uint32)(xferpoints[i+
        }
    }
}
// Perform view update operations
void changedirection() {
    double temp;

    for(unsigned int i = 0; i < J_NUM_PATCHES * J_NUM_CURVES * J_NUM_POIN
        temp = ((double *)patcharray)[i] * cos(rotatehoz) + ((doubl

        xferpoints[i] = ((((double *)patcharray)[i] * cos(rotatehoz) -
        xferpoints[i+1] = (((double *)patcharray)[i+1] * cos(rotateve
        xferpoints[i+2] = temp * cos(rotatevert) + ((double *)patchar
    }
}
```

```
// Set new view parameters
void engage() {

    for(int x = 0; x < J_SCREEN_WIDTH; x++) {
        for(int y = 0; y < J_SCREEN_HEIGHT; y++) {
                depthbuffer[x][y] = 255;
        }
    }
    for(unsigned int i = 0; i < J_NUM_PATCHES; i++) {
        drawcurve(controlpoints[i][0][0], controlpoints[i][0][1], co
        drawcurve(controlpoints[i][3][0], controlpoints[i][3][1], co
        drawcurve(controlpoints[i][0][0], controlpoints[i][1][0], co
        drawcurve(controlpoints[i][0][3], controlpoints[i][1][3], co
    }
    for(unsigned int i = 0; i < J_NUM_PATCHES; i++) {
        for(unsigned int j = 0; j < J_NUM_CURVES; j++) {

            drawline(controlpoints[i][j][0][0], controlpoints[i][
            drawline(controlpoints[i][j][1][0], controlpoints[i][
            drawline(controlpoints[i][j][2][0], controlpoints[i][
            drawline(controlpoints[i][j][0][0], controlpoints[i][

            drawline(controlpoints[i][0][j][0], controlpoints[i][
            drawline(controlpoints[i][1][j][0], controlpoints[i][
            drawline(controlpoints[i][2][j][0], controlpoints[i][
            drawline(controlpoints[i][0][j][0], controlpoints[i][
        }
    }
    for(unsigned int i = 0; i < J_NUM_PATCHES; i++) {
        for(unsigned int j = 0; j < J_NUM_CURVES; j++) {
            for(unsigned int k = 0; k < J_NUM_POINTS; k++) {
                setpixel(controlpoints[i][j][k][0], controlpo
            }
        }
    }
```

```
        }
}

// The grunt of the program, doing the forward step operations
void drawcurve(Sint32 pd[], Sint32 pc[], Sint32 pb[], Sint32 pa[]) {
        Sint64 A[3], B[3], C[3], D[3];
        Sint64 a[3], b[3], c[3], d[3];
        Sint64 bb[3], cc[3], dd[3];
        const Uint64 n = 4, N = 16;
        Sint64 temp1, temp2, temp3;

        for(unsigned int i = 0; i < 3; i++) {
                temp1 = pc[i] - pd[i];
                temp2 = pb[i] - pc[i];
                temp3 = temp2 - temp1;

                D[i] = pd[i];
                C[i] = 3 * temp1;
                B[i] = 3 * temp3;
                A[i] = (pa[i] - pb[i]) - temp2 - temp3;


                d[i] = D[i];
                c[i] = C[i]+(B[i]>>n)+((A[i]>>n)>>n);
                b[i] = 2*B[i]+((6*A[i])>>n);
                a[i] = (6*A[i])>>n;
        }

        int coords[3];
        for(double t = 0; t < 1; t+=0.001) {
                for(int co = 0; co < 3; co++) {
                        // Display the traditional Bezier method
//                      coords[co] = (int)(A[co]*t*t*t + B[co]*t*t + C[co]*t
                        // Display it in forward difference form by tradition
//                      coords[co] = (int)((a[co]<<(n+n+n))*((t*(t-1)*(t-2))/
                        }
                setpixel(coords[0], coords[1], coords[2], 255, 128, 128);
```

```
}
const unsigned int numsteps = 1 << n;
for(unsigned int step = 0; step < numsteps; step++) {
    setpixel(d[0] >> n, d[1] >> n, d[2] >> n, 255, 128, 128);
    for(unsigned int i = 0; i < 3; i++) {
        dd[i] = d[i] + (c[i] >> n);
        cc[i] = c[i] + (b[i] >> n);
        bb[i] = b[i] + a[i];
        d[i] = dd[i];
        c[i] = cc[i];
        b[i] = bb[i];
    }
}
}
```