A FLEXIBLE, SCALABLE APPROACH TO REAL-TIME GRAPHICS

Paul Anthony Shrubsole

A thesis submitted in partial fulfilment of the requirements of The Nottingham Trent University for the degree of Doctor of Philosophy

March 2000

Abstract

There have been several implementations of multiprocessor graphics systems in recent years that achieve high levels of realism at high performance. However, it is common for these implementations to be deficient either in their flexibility, so that they become application dependent, or in their scalability, resulting in a limited shelf life.

The aim of this research project is to design and investigate a graphics architecture that is flexible and scalable, whilst providing high quality output in real-time.

To achieve this, the thesis first focuses on high performance anti-aliasing techniques. New algorithms for performing texture anti-aliasing called texture potential mapping are presented which provide very high quality output at high performance and can easily be incorporated into multiprocessing environments.

The thesis then focuses on multi-processing architectures that can incorporate these advanced rendering algorithms in a flexible way whilst achieving scalable real-time performance. The latter part of the thesis presents a new architecture called the cellular array that embodies both flexibility and scalability in its design. Simulation experiments of the cellular array are presented that examine the behavior of this architecture for different applications and provides strategies for ensuring that the multi-processing environment is consistently well balanced.

Acknowledgements

The author expresses his sincere gratitude to Dr. Richard Cant for his expert guidance both in and outside of the field of computer graphics and also for his continued support during this program of study.

Gratitude also goes to Professor Andrzej Bargiela and also the other members of the Real Time Telemetry Systems Group who helped provide both an intellectual and friendly research environment.

I would also like to thank my girlfriend, Trudie van Kleef and friends and family for their help and support during the writing of this thesis.

Table of Contents

CHAPTER 1
REAL-TIME GRAPHICS ISSUES

1.1 Introduction

The realm of computer graphics has been studied and researched with intensity and vigour ever since the conception of the vector display in the nineteen sixties. With the rapid progress in speed, capacity and affordability of computer hardware over the past thirty years, however, researchers and industry are still subservient to the immense computational demands that are placed on computer systems in order to render realistic three-dimensional scenes in real-time. These pressures have forced programmers and hardware designers to consider computer architectures that make use of multiple processors in order to share out the objects or the tasks that render them in order to achieve both immersion and realism within a virtual reality environment. Furthermore, in order to make the most of available time, money and resources, computer graphics systems tend to meet needs for the technology of the day and thus lack of both flexibility and scalability. Such a philosophy, however, ultimately leads to systems with a very limited shelf life.

The thesis proposes and explores a new software/hardware architecture that can cater for the ever-growing demands on high performance and realism in real-time 3D computer graphics. The report is split into six chapters, each focusing on a particular aspect of real-time graphics that is constantly in need of improvement in terms of both visual quality and performance.

This chapter looks retrospectively at the evolution of real-time graphics systems in order to lay out the key problems that 3D graphics architectures have to deal with and with which this thesis aims to solve. Chapter 2 focuses on strategies for overcoming the effects of aliasing in computer graphics and provides a way of dealing with multiple pixel fragments that result from polygon rendering. This solution is then introduced again as a candidate scheme for the work in chapter five. Chapter 3 looks at texture mapping and presents two new algorithms for high quality anti-aliased texturing. Chapter 4 looks at multiprocessor

graphics architectures and examines their relative merits and disadvantages. A new graphics hardware architecture is then presented in Chapter 5 that offers flexibility with respect to types of graphics algorithms that may be employed and hardware scalability with respect to performance demands. Chapter 5 also incorporates all of the previous research illustrated in chapters two and three. The final chapter wraps up the research presented in the thesis and provides a set of recommendations for future research and development.

1.2 Real-time Image Generation

In order to understand the operation and ethos of current real-time graphics systems, it is valuable to look retrospectively at the evolution of real-time image generation. Since the early days of vector displays, which could draw only dozens to hundreds of lines in real-time [SUTH63], the following features were introduced for the first time in high-end, real-time 3D graphics systems:

- **Flat shading with lighting. 1977 [SCHA83]**
- **Gouraud shading and anti-aliasing of hundreds of polygons. 1977 [SCHA83]**
- **Gouraud Shading of thousands of polygons. 1988 [AKEL88]**
- **Antialiasing (on workstation). 1990 [HAEB90]**
- **Textures. 1988 [APGA88]**

Although earlier instances of the above have been implemented, (e.g. the Apollo landing simulator used flat shading with lighting in 1967) they were more one-off experimental systems.

Until 1988, the only systems that could generate anti-aliased images with textures in real-time were high-end military flight simulators costing millions of pounds. These systems had relatively low polygon performance compared to graphics workstations [SCHA83]. Graphics workstations traditionally have focused on displaying large numbers of primitives, which are necessary for computer-aided design and other modelling applications.

In the last several years, the two approaches have begun to converge. Flight simulators have increased their polygon performance and graphics workstations have begun to support antialiasing and texturing [EVAN91]. This is not particularly surprising since computer graphics has become much more commercially competitive in the past decade and certainly more accessible since the conception of 3D accelerator cards for the PC.

A successful graphics system for today's requirements and for those of the future should therefore provide both ***flexibility*** and ***scalability*** in order to maintain a high polygon

performance with realistic rendering. A ***flexible*** graphics architecture allows us to adapt and replace graphics geometry and rendering algorithms to suit the needs of specific applications. A ***scalable*** graphics architecture allows us to maintain an acceptable frame-rate as the demand for visual realism grows.

In order to see how these requirements can be met, we must break down the performance and realism issues into a specific set of requirements and objectives.

1.3 Objectives for Real-time Systems

Current systems display complex datasets consisting of tens to hundreds of thousands of primitives in real-time and support fairly realistic lighting and shading models. Future systems must exceed these performance levels, should be free of distracting artefacts, and should provide support for more realistic rendering. The following is a review of the most important issues that all graphics architectures need to deal with. The discussion also provides the basis for the research issues discussed in the later chapters (other graphical effects such as shadow generation, environment mapping, radiosity, etc., fall as sub-categories of those mentioned).

1.3.1 Screen Resolution

Ideally, a graphics system's display resolution should match the resolution of the human eye. Experiments on the human visual system indicate that the human eye can resolve features separated by 1 to 10 arc minutes, depending on the brightness and contrast of the features [ROSE73]. If we assume an 18-inch wide display screen viewed from 18 inches, this corresponds to a linear screen resolution of 350 to 1150 pixels. Current high-resolution monitors can comfortably display 1280 by 1024 pixels. This appears to be a reasonable standard for high performance image-generation systems (head-mounted displays *still* require significant improvements in this area and shutter glass technology appears to be currently superseding this technology).

1.3.2 Lighting and Shading

A graphics system must accurately model the interaction between light and the elements of a scene. The physics of light transport is fairly well understood. Unfortunately, to model the physics exactly for non-trivial scenes requires a prohibitive amount of computation. Rather than model the physics exactly, rendering algorithms make approximations that are less costly to compute.

The most realistic methods that are in common usage are ray tracing and radiosity. These are extremely compute-intensive, sometimes requiring days to compute a single scene

(radiosity has the advantage however in that surface interactions can be pre-computed for static scenes). Generally, the more realistic the lighting and shading model, the more computation that is required.

Most current high-performance systems support Gouraud or Phong lighting with Gouraud shading. A few systems support Phong shading. Future systems need to support Phong shading with multiple, local light sources in order to provide any significant boost in realism. Furthermore, future systems should be flexible enough to allow new shading algorithms to be added without affecting the overall graphics architecture. This is a non-trivial problem to solve if traditional interpolation techniques are employed throughout the graphics pipeline from frustum clipping to rasterization.

1.3.3 Anti-aliasing

Raster display systems, by their nature, cause the familiar problem of "jaggies" or "stair-casing". This artefact is a manifestation of a sampling error called aliasing in signal processing theory. Other common aliasing artefacts in computer graphics can be observed when mapping textures onto 3D surfaces and can lead to stomach curdling results during animations as the contents of textures swim around on the screen.
Aliasing artefacts occur when a function of a continuous variable that contains sharp changes in intensity is approximated with discrete samples. In order to minimise the errors that result from aliasing, a scheme of anti-aliasing should be introduced.
There are many strategies for overcoming aliasing (many of which are too compute intensive to justify real-time implementation) and these will be discussed in more detail in the next chapter.

Most high-performance systems support depth buffer methods in order to remove surfaces that are obscured by surfaces closer to the viewer. This inevitably leads to problems when polygons of varying depths contribute toward a single pixel and hence, aliasing effects are observed. Current and future systems must allow for special cases, such as polygon intersections and pixel contributions from several primitives without placing too much of a burden on processing. Furthermore, the scheme of anti-aliasing adopted should be algorithm neutral and not dictate the overall architecture of the system.

1.3.4 Frame Rate

To create the illusion of a moving image, the image must be updated rapidly. 13 Hz is an absolute lower limit for motion to appear smooth for fixed objects in a scene.

Even this is insufficient if the view or elements in the scene move rapidly. Flight simulators are one of the most demanding applications and require update rates of 30 to 72 Hz in order to avoid temporal aliasing. Frames must be double-buffered at least, so that only finished frames are presented to the user (unless the system is fast enough to refresh the screen ahead of the CRT beam). Another trick is to give objects the impression of movement by incorporating motion blur [POTM83] (itself computationally expensive) so that the frame rate can be lowered to an acceptable level.

Future 3D applications will employ more sophisticated rendering techniques. They will render scenes of greater complexity including objects consisting of many micro-polygons (smaller than one pixel). 3D graphics hardware that cannot achieve an acceptable frame rate for these new demands will obviously become obsolete.

1.3.5 Latency

Latency, the time between sampling user inputs and displaying the image, is a crucial issue for interactive, real-time systems [CANT96]. Users sense latency as a lag between movement of controls and a response in the visible image. Latency reduces controllability and the illusion of being immersed in the simulated environment. In certain applications, such as head-mounted displays or flight simulators, high latency can cause motion sickness.

Latency is an issue distinct from the update rate. Many high performance graphics systems increase their update rate by pipelining. For example, primitives of one frame can be transformed, while primitives from the preceding frame are rasterized. Pipelining increases the frame rate, but does not improve latency. The lower bound for latency is the frame update time. Flight simulators perform predictive tracking to minimise the apparent

latency, but user motions can not be predicted with complete accuracy, and therefore the results are not perfect.

1.3.6 Scene Complexity

The factor that generally receives the most attention in computer graphics research is the number of primitives that can be displayed per unit time. This determines the maximum complexity of the scene that can be displayed at a given rate. The highest performance systems available display in the order of millions of polygons per second at their peak rate. At a minimum update rate of 30 Hz, this corresponds to a scene complexity of the order of tens to hundreds of thousands of polygons. Such scenes do not approach the visual complexity of everyday life. Much higher performance is needed to display realistic scenes.

1.3.7 Texture Mapping

Rendering schemes that incorporate surface texturing greatly enhance realism but are computationally expensive. Current algorithms for texture mapping tend to cause blurring in order to anti-alias the textured surface with minimal computational cost. Future systems will need to produce better quality, high performance texturing that does not impede the remainder of the system [SHRU97]. This will be addressed in detail in chapter 3.

1.4 Compute Intensive Algorithms

The objectives set out above highlight that the most compute intensive elements of the rendering pipeline [MOLN90] involve the use of both anti-aliasing and texture mapping. Current algorithms that perform these tasks for real-time usage show deficiencies in either image quality or algorithm flexibility or both. Hence, a fair proportion of the research presented in the thesis has focused on developing new algorithms (or modifying existing ones) that meet the requirements of quality and flexibility whilst ensuring that computation is kept to a minimum.

1.5 Multiprocessor Architectures

Since the implicit requirements for the system involve both high performance and high scene complexity, we would like to process millions of polygons at update rates of 30Hz using anti-aliasing and texture mapping. However, the resultant number of computations per frame for such a requirement is too great for a single processor to handle. Thus, the rendering task must be distributed over multiple processors in order to achieve real-time performance. If care is not taken however, the design of the multi-processor architecture will place restrictions on the types of algorithms that may be used throughout graphics generation process and may also prevent expansion for additional processors by hard-wiring processor networks. Existing multiprocessing architectures, such as the PixelFlow system [MOLN92] are examined in chapter 4 along with their advantages and disadvantages. The remainder of the research is dedicated to exploring how the limitations on flexibility and scalability may be overcome by presenting a cellular array architecture that is flexible, scalable and makes use of the algorithms devised for texture mapping and anti-aliasing also presented.

CHAPTER 2
ANTI-ALIASING

2.1 Aliasing Effects in Real-time Computer Graphics

Traditionally, anti-aliasing techniques in real-time computer graphics have been considered a low priority when developing 3D graphics engines. This is understandable, since generally, there is no "quick-fix" to solve the problems of aliasing - solutions have tended to raise the cost of systems both in terms of resources and in terms of processing power. In the long term, however, (according to Moore's Laws), we find that such graphics systems have to be redesigned from scratch multiple times as new technological possibilities become feasible for real-time implementations. Thus, in order to design a *flexible* and *future-proof* multi-processor graphics architecture, we must ensure that anti-aliasing lies at its core. However, firstly we must investigate the causes of 'aliasing' in the context of computer graphics.

Aliasing is a common artefact within the realm of computer graphics that results from attempts to display a continuous geometric image containing high spatial frequencies on a discrete display device with a comparatively low sampling rate. High frequency components from the underlying image map incorrectly to lower frequencies, causing staircase stepping and moiré patterns in the static images. When a series of frames are generated in real-time, aliasing is even more apparent [SZAB83] and can lead to:

Classical temporal aliasing - like the backwards-spinning wagon wheel in old movies – fast speeds alias as slow speeds. Motion blur helps alleviate this problem.

Strobing - causing a fast moving object to appear to jump in discrete steps or appear stationary. This is similar to above problem but samples are now in-sync with cyclic variations in the signal.

Scintillation - small particles that drift between samples blinking on and off – scintillation of animated textures when texture mapping can lead to extreme artefacts (see next chapter).

The "**crawling ants effect**" - the spatial aliasing "jaggies" slowly changing between frames.

Stretching and shrinking - a slowly moving small object (only a few pixels in size) will seem to stretch and shrink in one-pixel steps as it crawls across the screen.

Dealing with these artefacts in an efficient and aesthetic way is not a trivial task and many attempts have been made to overcome the problem of aliasing in real-time 3D graphics. However, as mentioned earlier, many of these algorithms tend to lack flexibility since they are optimised for a particular environment in a bespoke fashion, or they lack the speed required to render a high number of texture mapped polygons in real-time. This chapter provides a taxonomy of the more established techniques of anti-aliasing for real-time usage and discusses possible strategies that have been implemented to deal with special cases of aliasing artefacts that are difficult to manage. Ultimately, we must establish the most suitable anti-aliasing scheme for a flexible, scalable multi-processing graphics environment and this is presented in the final section of the chapter.

2.2 Strategies for dealing with Aliasing

The aim of any antialiasing scheme is to attenuate frequency components in the underlying image that are higher than the Nyquist frequency of the display device. Any frequency component of the signal that is higher than this limit will be reconstructed as a lower frequency alias. Classical signal processing models using sampling and Fourier theory require a four-stage process consisting of generating and then passing an analogue signal a sampling system (e.g. an analogue to digital converter). Digital information is then processed and finally reconstructed (e.g. using a digital to analogue converter). All computer graphics anti-aliasing schemes must therefore band-limit the source before reconstructing the final image. This is generally easier to do in the Fourier domain by means of convolution [OPPE75].

By its very nature, antialiasing is a compute-intensive process, no matter which scheme is used. Most current systems that perform antialiasing do so with steep performance penalties (in some systems such as flight simulators, antialiasing is a fundamental part of the application; even in systems such as these, where antialiasing cannot be "turned off", it consumes significant hardware resources that could have been spent elsewhere). Antialiasing plays a large role in determining the factors involving realism and speed for real-time graphics hardware systems. Furthermore, antialiasing should not put too much load on the system since we must also consider other compute intensive tasks such as texture mapping, lighting and shading algorithms.

Antialiasing methods can be divided into two classes: ***analytic*** methods, which can pre-filter an image and take out its high frequencies before sampling pixel values, and ***discrete*** methods, which estimate pixel coverage by sub-sampling using a brute force approach. Discrete anti-aliasing algorithms have historically been chosen in preference over analytic methods since they may be incorporated into systems that have initially been designed without thought of anti-aliasing. However, the analytic approach is becoming more popular in commercial implementations since it has become an algorithmically rich area of study with realistic solutions for real-time graphics. Each of these schemes will now be discussed in more detail.

2.3 Discrete Anti-aliasing - Supersampling

Many hardware and software systems anti-alias using some form of supersampling. This involves sampling the image at higher than pixel resolutions and subsequently filtering the samples down to a single sample per pixel. Supersampling can be done adaptively, based on the local scene complexity of an image [WHIT80], but its simplest and most regular form involves over-sampling the image uniformly such that an intermediate image is computed at a higher resolution than the final output. Combining several of the pixels in the high-resolution intermediate image then forms the output pixels. Combining intermediate pixels, which overlap the output pixel over a range, by using a weighted average can provide good quality results.

Although supersampling appears trivial, it is really a combined implementation of three conceptual steps, although algorithmically the second and third stages are combined:

(i) Oversampling

High-resolution digital image – a continuous image is sampled at *n* times the final resolution to produce a virtual image.

(ii) Digital low-pass filter

Band-limited high-resolution digital image – the virtual image is lowpass filtered.

(iii) Decimation

Low-resolution digital image – the filtered image is resampled at the final frame store resolution.

Figure 2.1 illustrates the distinction between classical anti-aliasing (Figure 2.1a) and supersampling. Figure 2.1b shows that no channel between analogue information and a sampler exists and therefore no point in which an anti-aliasing filter can be inserted in order to band-limit the image. As continuous data is sent from the scene database (in most real-time cases, polygon hierarchies), the rendering system interpolates along polygon edges in screen space and samples discrete values. Since there is no continuous function that can be sampled anymore, we simply increase the rate of sampling in order to reduce aliasing artefacts.

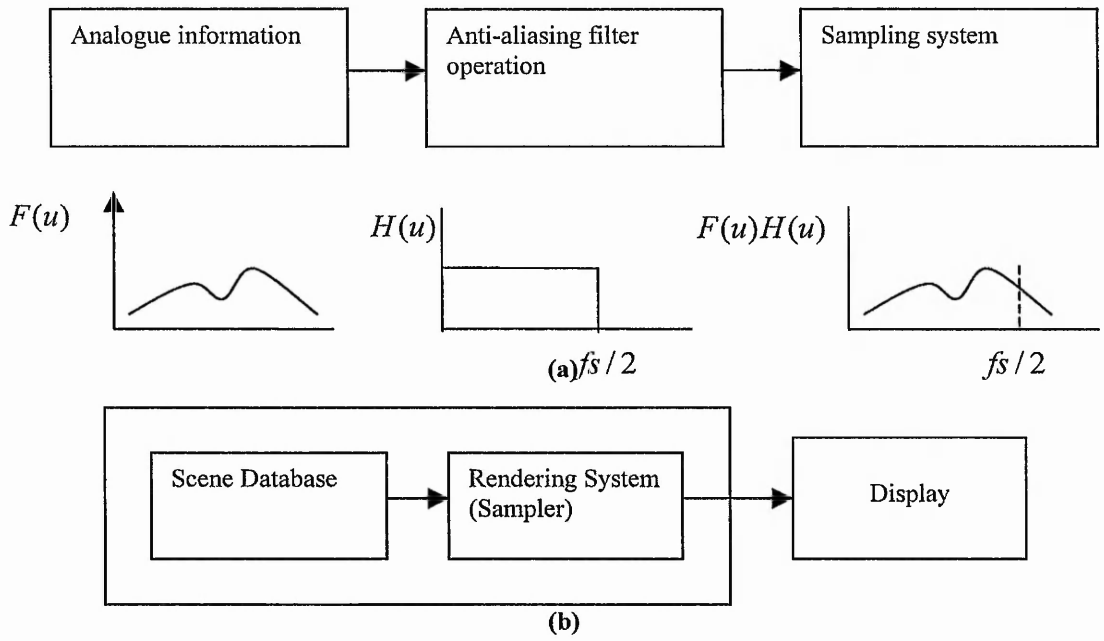


Figure 2.1 **(a) The 'classical' approach to anti-aliasing, (b) A graphics generation system using discrete sampling.**

Practical approaches to supersampling involve sampling multiple images at the resolution of the final display with each one being displaced by a fractional unit of the filter kernel size being applied (e.g. a 3x3 box filter results in 8 'displaced' virtual images). These results are then summating per pixel and dividing the size of the kernel.

Supersampling can be very expensive if a large number of samples are used to form each output pixel. In order to overcome this problem, stochastic supersampling was developed in order to reduce the number of samples by randomising the sampling process and scattering high frequency information into noise [MITC87].

2.3.1 Advantages and Disadvantages of Supersampling methodologies

Supersampling methods are very popular in graphics implementations and for good reason. The supersampling algorithms themselves are easier to implement than analytic methods. Furthermore, it is a relatively simple task to 'bolt' supersampling techniques onto an existing rendering scheme. This is an attractive gain in terms of flexibility. However, several disadvantages to supersampling inevitably make it less attractive.

The most obvious technical drawback is that there is both an economic and technical limit to increasing the resolution of the virtual image. Supersampling methods are essentially *brute force* methods, which in turn, imply brute force costs. The computational power required for a renderer is roughly proportional to:

$$\textit{No. of primitives} \times \textit{No. of subpixel samples} \times \textit{Frame rate} \times \textit{Primitive size}$$

If we increase the number of samples by a factor of k, we can increase the renderer's power by a factor of k, reduce the number of object primitives, or reduce the frame rate, which is undesirable.

A further problem involved with making use of supersampling for real-time systems is the immense bandwidth required. If we consider a system that produces real-time 3-D images at a frame rate of 30 frames per second on a screen of resolution 1280 * 1024 and a supersampling density of 8 samples per pixel, we would require a bandwidth of:

$$1280 \times 1024 \times 8 \textit{ samples/pixel} \times 3 \textit{ bytes/sample} \times 30 \textit{ frames/sec} = 0.94 \textit{ Gigabytes/sec}$$

Another more subtle drawback is that since the frequency spectrum of computer graphics images can extend to infinity, increasing the sampling frequency does not necessarily solve the problem, it merely *reduces* the aliasing by raising the Nyquist limit. In effect, supersampling simply shifts the effect up the frequency spectrum.

Ultimately, as well as flexibility and scalability, the following factors need to be considered for real-time interactive systems: image quality, bandwidth requirements and hardware complexity. Supersampling can produce very good results if implemented properly,

provided of course that sufficient samples are taken per pixel. However, the number of samples directly affects the bandwidth once we begin to consider the other algorithmic requirements of the system, such as hidden surface removal. It is therefore deemed necessary to consider other methods of antialiasing that are algorithmically inexpensive and yet produce good results with low bandwidth.

2.4 Hidden Surface Removal and Anti-aliasing

There are many strategies for removing hidden surfaces in real time 3D computer graphics. The most commonly used of these is the z buffer (or depth buffer) [CATM74]. For each pixel in the display, we keep a record of the depth of the primitive in the scene that is closest to the viewer, plus a record of the colour or intensity that should be displayed to show the object. When a new polygon is to be processed, a z-value and intensity value is calculated for each pixel that lies within the boundary of the polygon. If the z-value at a pixel indicates that the polygon is closer to the viewer than the z-value in the depth-buffer, the z-value and the intensity values recorded in the buffers are replaced by the polygon's values. After processing all polygons, the resulting intensity buffer can be displayed. This strategy has the advantage in that it is algorithmically simple and its only real overhead is its requirement for an extra block of memory of the order of the pixel resolution of the screen. Care must be taken, however, as to the range and accuracy of depth values as they are stored since a large range between front and rear camera clipping planes may result in ambiguous depth comparisons for points that are very close together.

A scan-line based rendering approach is generally employed within polygon edge spans in which z-values are interpolated. Such spans however will consist of a range of partially obscured pixels from the left and right edges and a single depth sample will be insufficient to avoid 'stair-casing'.

When antialiasing and hidden surface removal are combined, there could well be many such partial contributions from different edges within a given screen pixel depending on the depth complexity and relative screen polygon size. Analysing this problem geometrically leads us into the domain of continuous anti-aliasing.

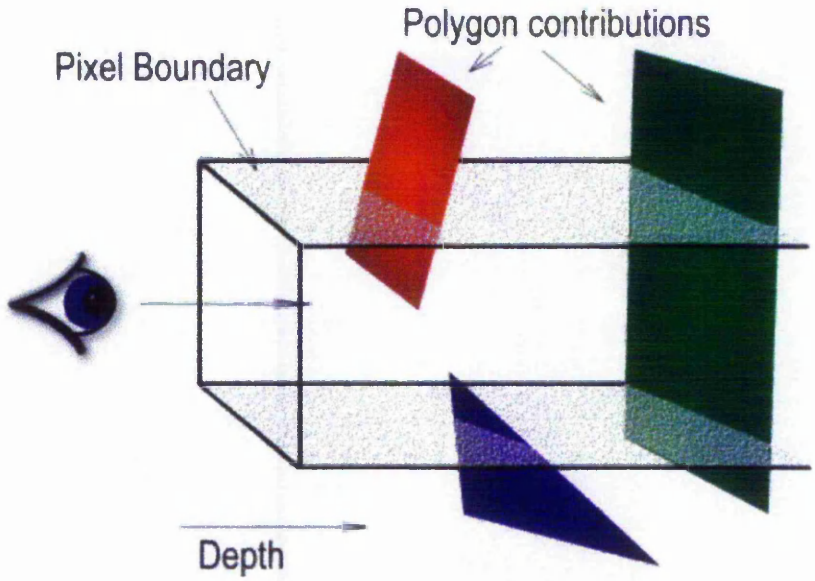


Figure 2.2 **There may be many contributions of polygon faces in a pixel that need to be taken into account for correct anti-aliasing**

As Figure 2.2 shows, there may be many contributions from polygon edges. However, the basic implementation of the z-buffer will only take the contribution from the nearest polygon and will disregard the remaining polygons. This approach will clearly produce spurious results and will be quite noticeable to the viewer when a large contribution to a pixel from an edge that is further away is disregarded.

A further problem arises when two polygons intersect each other as shown in Figure 2.3. In this case, there are likely to be many pixels where the two faces "overlap" in depth. A simple-minded approach to the depth comparison will not provide the correct answer. Again, only the contribution from the nearest face will be provided. Pixels where intersecting surfaces are visible usually number in the hundreds in a typical 1280x1024-resolution picture.

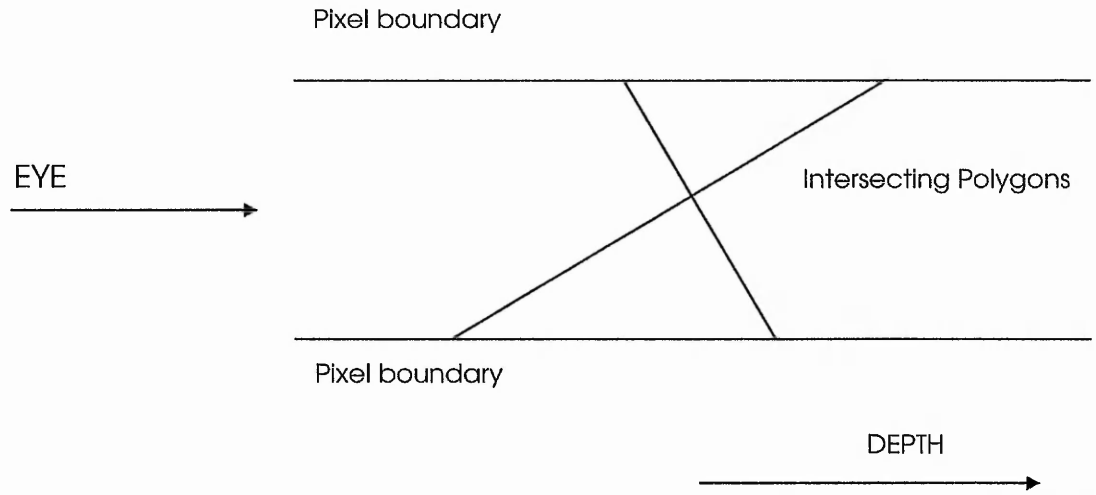


Figure 2.3 **Two polygon faces interpenetrate so that two contributions are made to the pixel.**

In order to deal with these problems directly, we must work in the continuous image domain and process multiple polygon edge contributions for each pixel in screen space geometrically. The A-buffer provides an elegant solution to this problem.

2.4.1 Analytic Algorithms – The A-buffer

In order to examine possible solutions to the problems of aliasing, it is relevant to categorise the main sources of high frequency components in computer-generated images (with the exception of texture mapping):

Explicit Edges: The zero-width boundaries of primitives have infinite spatial frequencies.

Implicit Edges: Interpenetrating object primitives cause implicit edges, which also have infinite spatial frequencies.

High frequency Shading Effects: Certain shading effects, such as specular highlights, shadows, and textures can contain high frequency components. For example, a Phong-shades cylinder several pixels wide can have a specular highlight much narrower than a pixel.

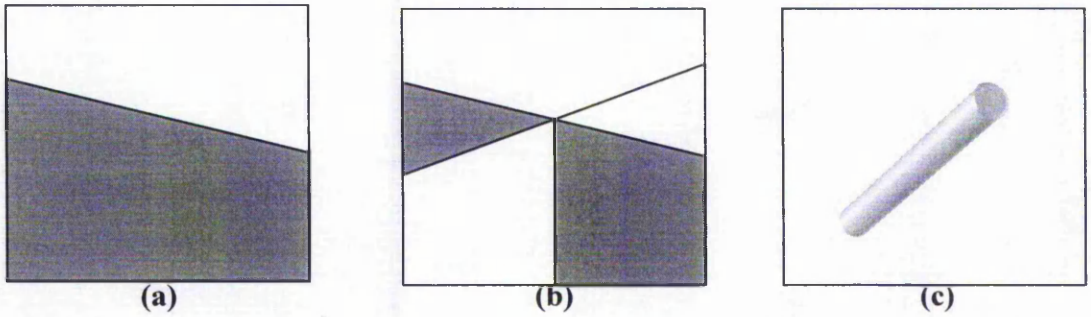


Figure 2.4 **Sources of high frequency components in a displayed image: (a) Explicit polygon edge (b) Implicit edge where primitives intersect and (c) specular highlight.**

If we ignore the process of texturing objects (see next chapter), we find that the main source of high frequency components in most images is along polygon edges. Pixels near polygon edges need extra sampling; pixels in the interior of polygons can generally be sampled once per pixel. It is practically impossible to pre-filter intensities across polygons exactly [WATT98] (with the exception of flat shaded polygons) since variations in intensity are analytically too complex. However, we can evaluate fragment geometry to any desired level of accuracy and use approximation to disctretise intensities.

A-buffer algorithms (anti-aliased, area-averaged, accumulation buffer) provide a hybrid solution to analytic and discrete anti-aliasing by storing sub-pixel fragment information for pixels that contain polygon edges, and simply storing colour and depth values for the others. Fragment information can then be combined with polygon edges that further contribute to a given pixel as the scene database is being processed. Fragment geometry is straightforward to calculate analytically when object primitives are only polygonal and is normally discretised using bitmasks.

The original A-buffer algorithm was used to produce high quality anti-aliased images in the initial Reyes rendering system [CARP84]. This system was designed to work on virtual-memory uniprocessors, such as the VAX 11/780. The algorithm, described by Carpenter assumes that rendering is performed using a screen sized image buffer. The entry for each pixel can contain a single colour and depth value, or else a pointer to a linked listed of partially covering primitive surfaces. Initially, each pixel in the screen buffer is set to the colour and depth value of the background. As primitives are rasterized, they are diced into

pixel sized pieces call fragments. Fragments may completely cover a pixel, or may only partially cover it (if an edge of the polygon passes through the pixel, for example).

At each pixel affected by a polygon, a determination is made as to whether the fragment covers the pixel completely or partially. If it covers the pixel completely and is the closest surface so far, the fragment's colour and depth value replace the values stored in the screen buffer. If it only covers the pixel partially, a record is made containing the fragment's colour, depth, and partial coverage information (generally an alpha value).

The entry in the screen buffer is changed to a pointer, which points to this newly created record. This record contains a pointer to another such record, containing the colour and depth of the background. As additional primitives are rasterized, they are merged into the screen buffer in this fashion.

In the original A-buffer paper, there is no restriction on the length of the linked lists of fragments. In pathological cases, where many primitives partially cover a single pixel, the linked lists can become arbitrarily long. However, fragments that completely cover a pixel obscure all of the fragments behind them and truncate the list. This restricts the length that lists can actually achieve. The structure of the list contains the following elements:

- **Red, green and blue colour components**
- **Coverage (fractional double precision number representing area coverage)**
- **Object Tag**
- **Bitmask (size should be user defined for control of quality)**
- **Z depth**
- **Fragment Pointer**

The fragment pointer maintains a link with other facets that are found to contribute to a given pixel so that they may be processed at a later stage. The pixel coverage gives an accurate value of the fraction of the pixel that is being obscured. One way to obtain the area coverage is by assuming a circular pixel and using the perpendicular length from the circle centre to the polygon edge as a key to a lookup-table. This method provides considerable efficiency gains.

Bitmasks provide an efficient means of comparing the overlap between pixel fragments [ABRA85]. We can thus combine the contributions of these fragments by performing logical XOR operations on each successive pixel fragment that partially obscures the previous. Achieving a Bitmask representation of a primitive edge intersection can be done by finding where the edge intersects with the pixel edges, and then using a lookup table to determine a 16-bit word representing "on-off" states in a 4x4 grid. Thus, Bitmask subpixels are "on" if they lie within or on an edge and "off" outside the edge.

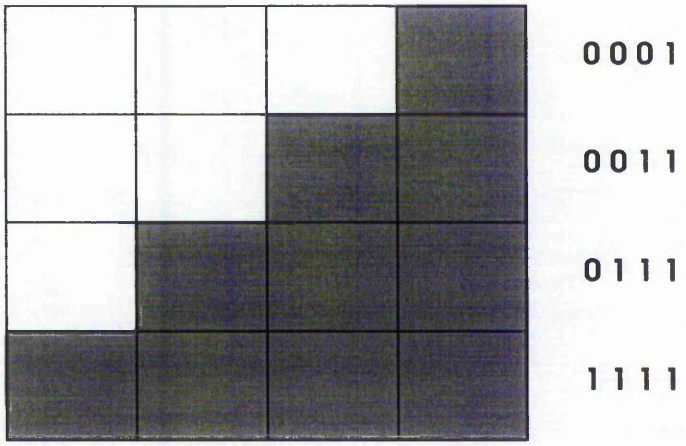


Figure 2.5 **Polygon pixel fragments are represented using masks of bits.**

Figure 2.5 shows how the Bitmask can be implemented. The edge intersection points are determined by linear interpolation and require minimal computation. The resultant Bitmask value in this case would be the 16-bit word 0001001101111111.

An index that identifies which object that is being considered (called an object tag) is also stored in the list structure. This value can then be used to save time when two faces that belong to the same object intersect. Since the faces of the edges will meet perfectly, we can simply perform an OR operation without worrying about obscuring the pixel partially. This requires a slight modification to the polygon database to cater for this feature and does not incur any significant increase in complexity of the algorithm.

One version of the A-buffer implemented during research studies avoids the use of a variable list structure for fragment comparison altogether, and makes use of a fixed set of buffers instead. This avoids a possible loss in performance due to fragmentation of the data in memory. It also helps to provide more control on the upper limit of depth complexity (i.e the number of fragments per pixel) in the scene.

Experiments using the multiple A-buffer scheme exhibit anomalies in scenes which display objects lying *on* planar surfaces. These anomalies can be observed along the edges of objects that make contact with the surface. A more stringent algorithm is therefore required in order to take special cases of edge intersections into account.

2.4.2 Overlapping Pixel Fragments

Planar intersection is assumed if the depth ranges of two different objects overlap. Figure 2.6 shows how that orientation of the objects is assumed and the ranges with which to determine overlap.

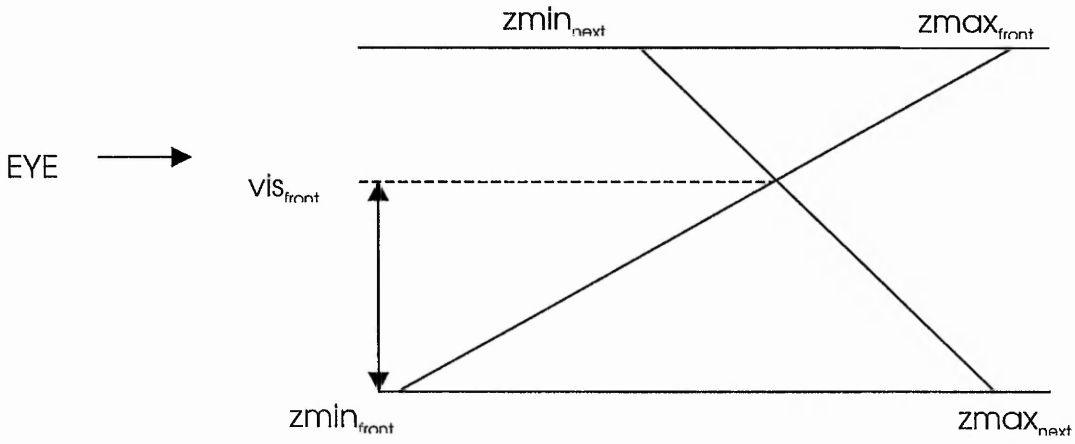


Figure 2.6 **Finding the extremes of the edge intersections is important in order to detect and evaluate overlap contributions to a pixel (vis_{front}).**

Once overlap has been established, the visible area of the front fragment can be determined using the following equation:

$$\mathbf{vis}_{front} = \frac{\mathbf{zmax}_{next} - \mathbf{zmin}_{front}}{(\mathbf{zmax} - \mathbf{zmin})_{front} + (\mathbf{zmax} - \mathbf{zmin})_{next}}$$

However, there are certain instances where the calculation fails to produce a correct result. Since only one proper depth sample is taken for each pixel at its centre, erroneous values will occur when the pixel centre does not lie inside of the polygon edge. This could occur when objects within a scene lie on a surface such as a tabletop and samples are taken around the edge from the table, rather than from the object. Figure 2.7 illustrates this problem, where the depth sample for face one lies outside of the object and hence the z-slopes for this face cannot be determined. Thus, the contribution from face one is ignored and only face two is considered.

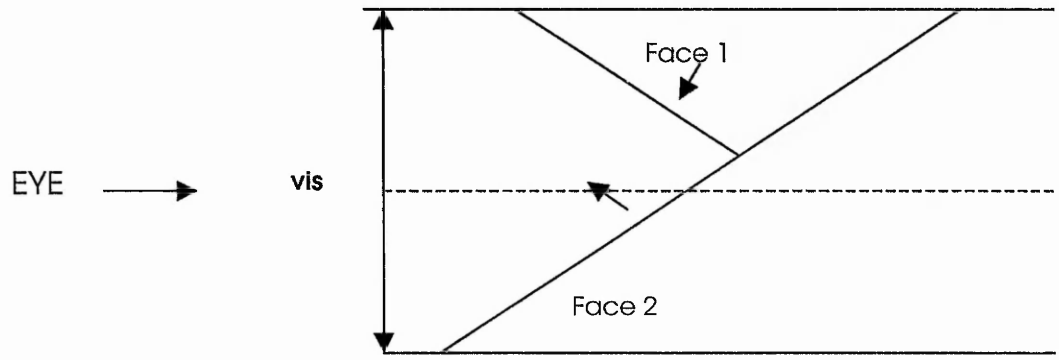


Figure 2.7 **Face 1 is not taken into account since sampling only takes place at pixel centres.**

We can minimise depth misses by simply taking more z samples (the discrete approach). Figure 2.8 shows the case when additional z samples are taken at the pixel corners. Depth misses, although less likely, *will* still occur. Essentially, we face the same problems discussed earlier in the chapter regarding supersampling. A better approach therefore, is to retain continuous geometric information about pixel fragments.

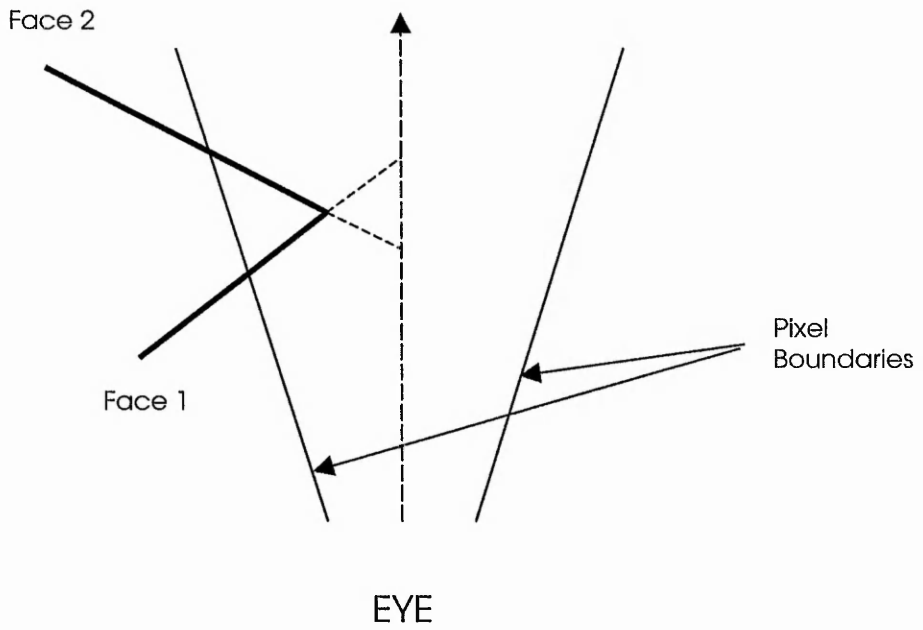


Figure 2.8 **We can take more z samples to avoid depth misses that lie outside of polygon boundaries but we are minimising the problem rather than solving it.**

Several strategies have been investigated in order to gain a more accurate means of establishing depth values within pixel fragments. The most successful of these involves storing both the slope of z with respect to screen-space x (dz/dx) and the slope of z with respect to screen-space y (dz/dy) along the pixel centre depth value into the buffer. Hence the calculations that are required at points within the pixel can be readily interpolated for use with the Bitmask. Such an implementation produces better results. Storage-wise, the enhancement will incur an additional 8 bytes per pixel in order to use the floating point slope data along with the fragment data effectively.

2.4.3 The Priority Mask

Schilling [SCHIL93] proposed an enhanced solution to the above problem by generating a subpixel mask that indicates which part of the pixel object is the front object and in which part of the pixel the other object is the front object. This subpixel mask can be used to modify the edge subpixel masks of the two objects so that:

$\mathbf{A_{new} = A \ \& \ NOT \ (A \ \& \ B \ \& \ NOT \ (C))}$ and

$B_{new} = B \ \& \ NOT \ (A \ \& \ B \ \& \ C)$

Where A is the edge subpixel mask for the first object; B is the edge subpixel mask for the second object and C is the mask for both objects, known as the priority mask.

The calculation of the priority mask uses the increments for the depth value in the x and y directions in order to define a plane that indicates where the first plane of the first object is in front of the second plane by the sign of its z-value. The intersection with the plane $z = 0$ denotes the border between the two areas where the first plane or the second plane respectively is in front of the other plane. The representation of this plane looks similar to the representation of the polygon edges in some rendering systems such as the pixel planes system (this will be described in more detail in chapter 4). The mechanisms that exist to generate subpixel masks representing edges can therefore be used to generate the priority mask.

We also need to test for cases in which a p-mask calculation is required since performing these p-mask calculations on non-overlapping fragments is a rather redundant exercise. The criteria are:

(i) A & B does not equal 0 (in which the subpixel masks of the two objects do not overlap)

(ii) $z2 - z1 < (|dz2,x - dz1,x| + |dz2,y - dz1,y|) / 2$

The second criterion ensures that the case in which the intersection of the two objects occurs within a pixel area is true.

In order to increase efficiency, a lookup table can be used to find the p-mask based on the line of intersection of two planes and based on the depth and difference in depth with respect to x and y. The parameters are used for use with a lookup table using the straight-line equation (origin of co-ordinate system at pixel centres).

$$F(x,y) = z + x*dz_x + y*dz_y = 0$$

The biggest difference in implementation between the A-buffer and traditional z-buffer techniques is that the A-buffer contains lists of contributions to each pixel whereas the z-buffer only stores one item per pixel (i.e. the one that is closest to the viewer). Many recent rendering systems employ the z-buffer simply because the task of handling lists is deemed too expensive to implement in hardware.

2.5 Alternative Strategies for Filtering in the Continuous Domain

An ideal pixel fragment composition algorithm would be one that could recursively refine the accuracy for determining pixel contributions to *any* depth. In this case, the level of anti-aliasing could be varied according to the user's priorities. It turns out that the hidden surface removal algorithm developed by Warnock [WARN69] can meet this requirement with some modification since it can, in theory, recursively look at primitives down to subpixel resolution. Although the algorithm is elegantly simple, it consumes a considerable amount of processor cycles compared with the A-Buffer. The algorithm therefore, is not really considered as a serious contender and is included for the sake of intellectual completeness.

Warnock's algorithm belongs to a class of area-subdivision algorithms that follow the *divide and conquer* strategy of spatial partitioning in screen space. This is achieved by firstly dividing the screen into four quadrants. Polygons are displayed if a straightforward decision can be made as to whether they are visible in a given area. If this condition is not met, then the area is recursively subdivided into smaller areas until a decision can be made. The projection of each polygon has one of four relationships to the area of interest as illustrated by figure 2.9.

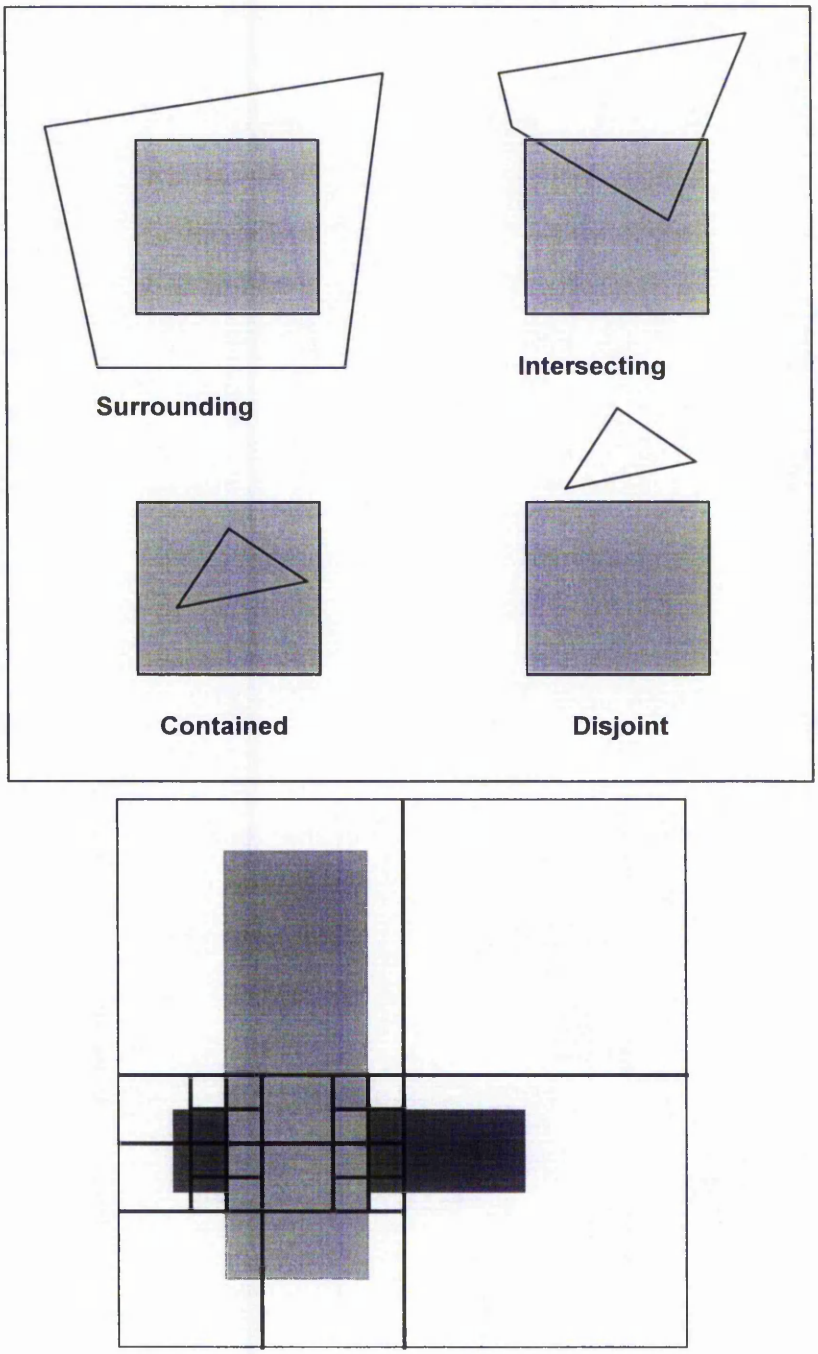


Figure 2.9 **Top: The four possible relations of projected polygons to an area square area element. Bottom: The black lines show how the image space has been divided up using this algorithm.**

Traditional use of this algorithm ensures a cut-off point for the recursive reduction procedure when a resolution of one pixel is reached. However, we can exploit this scheme for situations where there are several sub-pixel contributions from different faces. It would

seem logical to continue reducing the area elements in the same way, until a satisfactory level of sampling is achieved.

A modified Warnock's algorithm appears to be an attractive alternative to the A-Buffer since hidden surface removal and anti-aliasing would be provided by the same algorithm. Due to the recursive nature of the reduction algorithm, it would also be feasible to implement in hardware. However, the algorithm also opens up opportunities for worst-case scenarios where performance will drop significantly. This is the result of the number of operations being based on an order of *O(pixels x polygons x sub-pixel-resolution)*.

2.6 Recommended Scheme for Hardware Architecture

Discrete anti-aliasing methods, such as supersampling, provide only an incomplete solution to the aliasing problem. An easy classical solution is not available because unlike image processing, a continuous image-to-sampler channel does not exist. For example, a ray-tracer intertwines the both the generation process and the sampling process together.

The alternative however, is to pre-filter an image and take out its high frequencies *before* sampling the pixel values. In this case, we must examine the geometry of object primitives before they are rendered. Bearing in mind that we can only approximate the required filter (area calculations can be precise but intensities are discretised), we still have much more control between qualitative results and efficiency. We therefore conclude that filtering in the continuous domain provides a more robust solution in which a special-purpose analytic render must be designed.

The modified A-buffer provides a well-balanced solution to continuous filtering in that it provides the most balanced compromise between performance and quality using geometrically derived sub-pixel fragments. Thus, the A-buffer is chosen as the candidate anti-aliasing scheme for the proposed multi-processor architecture and will be re-introduced in chapter 5.

Chapter 3

Texture Mapping

3.1 Motivations for Texture Mapping

Early real-time graphics systems, such as flight simulators, gave us an insight into the practicality of interacting with 3-D environments. Their main criticism however was their lack of realism due to the extreme smoothness of surfaces. Not only were the scenes visually uninteresting, but it was found that pilots training on flight simulators were unable to make use of visual motion cues when "flying" at low altitude. However, as greater processing power became more readily available, methods were devised try to give a better representation of surfaces, thereby adding richness to the scene and linking more closely to their "real-life" counterparts. The most startling difference can be observed by overlaying a predetermined texture to each surface (see figure 3.1). Two aspects of texture that are usually considered are the addition of a pre-specified pattern to a smooth surface and the addition of the appearance of roughness to the surface. Thus, the demand for graphical realism in modern real-time graphics systems has meant that texture mapping has become a vital component in the graphics-rendering pipeline. However, the process of accurately mapping texture values to the display requires a significant amount of computing power.

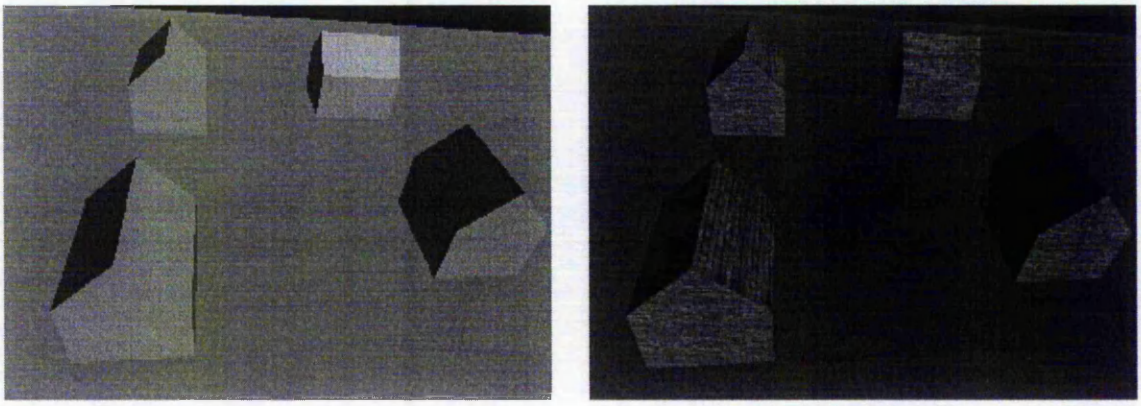


Figure 3.1 **A simple rendered scene using flat shading (left) and the same scene using texture mapping (right).**

Normal real-time image rendering methods require that instead of mapping the texture values (texels) on a polygon in 3-D space to the screen, screen pixels are effectively mapped onto a region of the texture using a reverse projection. This means that whilst image results would be fine when transformed screen pixels are of the same order of size, shape and orientation as the stored texture pixels, when the sizes of the transformed pixels in texture space vary, aliasing can readily occur causing the resulting texture of the image to swim and scintillate. Therefore, a method of finding the average texture value within the pixels needs to be adopted. Various techniques of dealing with aliasing in real-time systems have been established over the years [HECK88]. All of them try to minimise its effects by using various degrees of approximation. These methods however, avoid the problem of directly dealing with general quadrilaterals in texture space, and will fail to produce pleasing results at certain viewing distances and orientations. The result in this case will be over-blurring or remnants of aliasing artefacts.

An alternative technique is therefore desirable in order to improve the quality of texture mapping results at high-performance without requiring excessive computational power.

3.2 The Mapping Process

Since the basis of adding planar texture patterns to smooth surfaces is mapping, the texture problem reduces to the specification of a transformation from one co-ordinate system to another. The most commonly used method is to scan in screen space (x,y) and find the transformation that maps to texture space u(x,y) and v(x,y). Thus, the amount of work done during a point-to-point transformation for a polygon in object space is directly proportional to the number of pixels the polygon covers in screen space

The mapping from screen space to texture space can be deduced by applying a perspective matrix transformation expressed in homogeneous form (where division by w and q are required for transforming to perspective view co-ordinates):

$$[xw\ yw\ w] = [u\ v\ 1] \begin{pmatrix} A & D & G \\ B & E & H \\ C & F & I \end{pmatrix}$$

Hence the values of u and v are evaluated by finding the inverse of this matrix:

$$[uq\ vq\ q] = [x\ y\ 1] \begin{pmatrix} EI-FH & FG-DI & DH-EG \\ CH-BI & AI-CG & BG-AH \\ BF-CE & CD-AF & AE-BD \end{pmatrix} = \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

Resulting in:

$$u = \frac{ax + by + c}{gh + hy + I} \qquad \text{and } v = \frac{dx + ey + f}{gh + hy + I}$$

Since the formulae for u and v are quotients of linear expressions, uq, uv and q can be computed incrementally when using scan-line methods for rasterization. This further reduces the computational cost to 3 additions and two divides per screen pixel.

3.2.1 Other Texture Parameterisation Techniques

The affine transformation from camera space (i.e. after an inverse perspective transform from screen space) to texture space is derived from basis vectors whose u and v directions lie in the plane of a given polygon and whose origin lies at a vertex of the polygon. This results in a parameterisation defined by a translation and rotation and works well for large polygon surfaces.

However, u and v coordinates may also be defined by the geometry of groups of polygon primitives defining an object at the modelling stage. In this case, a three stage mapping process is generally used in which screen space maps to world space, object space maps to an intermediate three dimensional texture space and is finally mapped to two dimensional u,v coordinates.

3.3 Aliasing in Texture Mapping

If the method described above is used on a point-to-point basis, i.e. the centres of each of the screen pixels are mapped to single texture values, then large chunks of the original texture can easily become left out. This is analogous to the classic "jaggies" aliasing effect, where edges within a pixel are not sufficiently sampled (i.e. sampling takes place below the Nyquist rate). Texture aliasing arising from low sampling rates however, can often deteriorate picture quality more seriously than "jaggies" do since more pixels are affected. Furthermore, in a real time system, textures will tend to map at different orientations for each frame at a rate of 30 to 60 frames per second. The resulting texture may well exhibit Moiré fringes and will appear to swim and scintillate [DUDG91].

Such a problem can be dealt with effectively by making use of a procedural texture [SCHA80]. With a procedural texture, it may be possible to evaluate the average texture within a pixel analytically - but there are few choices of "texture function" which allow this - the major option being patterns built from a superposition of sinusoids (see figure 3.2). With such a system, exact sampling of texture is possible, resulting in high quality anti-aliased images. Some early flight simulator texture mapping systems utilised this approach. The drawback of this method is that the range of possible patterns is severely limited and hence it is not possible to reproduce the majority of textures found in the real world. Figure 3.2a shows the results of aliasing using of point-to-point mapping. When compared with the outcome of mapping procedural texture analytically, figure 3.2b, we see that anti-aliasing is clearly an essential part of the texture mapping process.
Aliasing can also be reduced by taking a large number of texture samples within a pixel and applying a filter in texture space [FEIB80]. Figure 3.2c makes use of 16 samples within each pixel evaluated incrementally and figure 3.2d uses 16 random samples for each pixel. Note that the latter method produces better results but still exhibits spurious areas of texture.

The method of applying more samples drastically increases the processing time required to render each surface in a 3-D scene. More subtle techniques are therefore required that retain as much of the original texture as possible whilst keeping the sampling rate low.

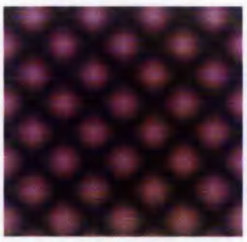


Figure 3.2 **Original source texture is mathematically derived for procedural texture mapping. The viewing angles for the subsequent textured surfaces are chosen such that the orientation of transformed pixels on an infinite plane produce long, thin footprints at 45°.**

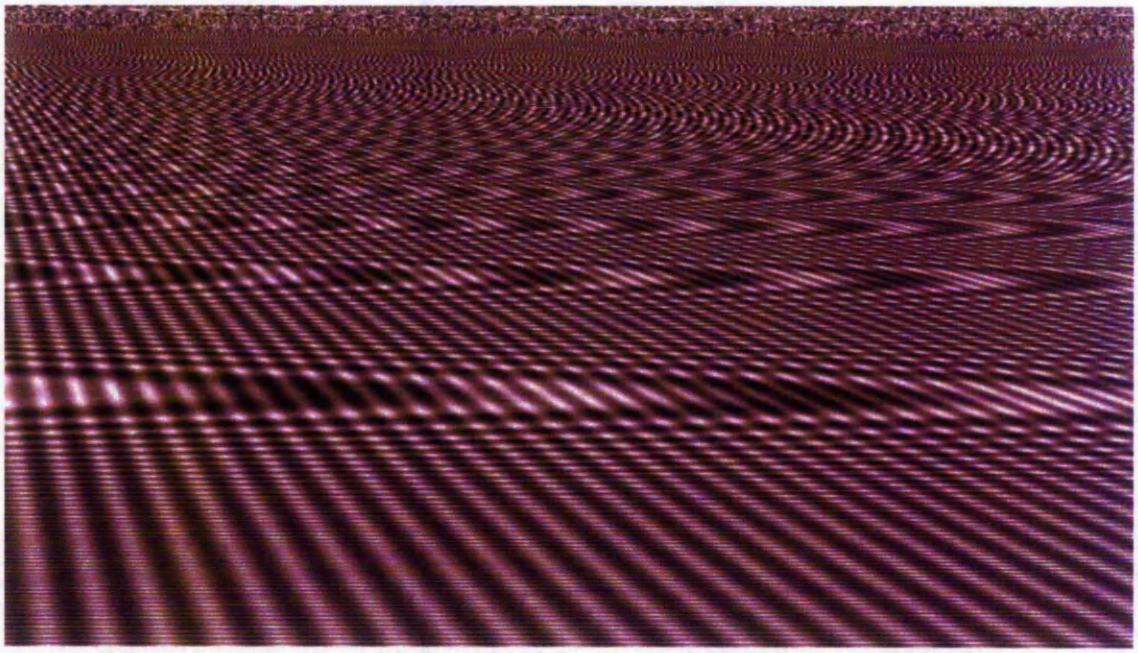


Figure 3.2 **(a) Texture is mapped on a point to point basis. Extreme aliasing effects and fringing are exhibited as the texture becomes compressed with distance.**

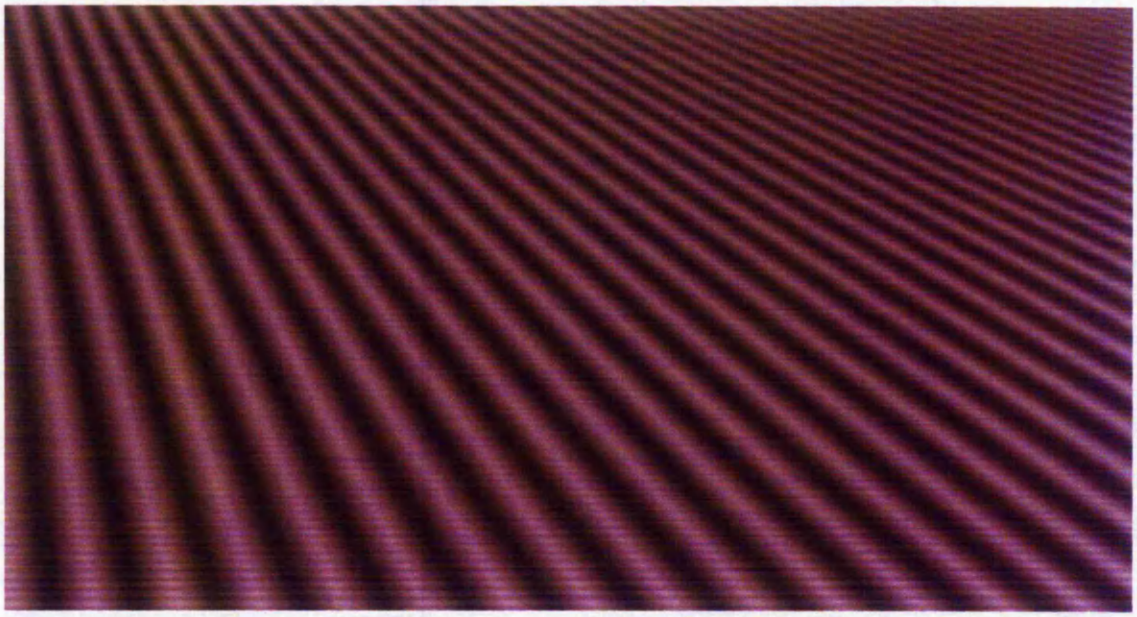


Figure 3.2 **(b) The procedural texture is mapped analytically, producing excellent anti-aliased results**

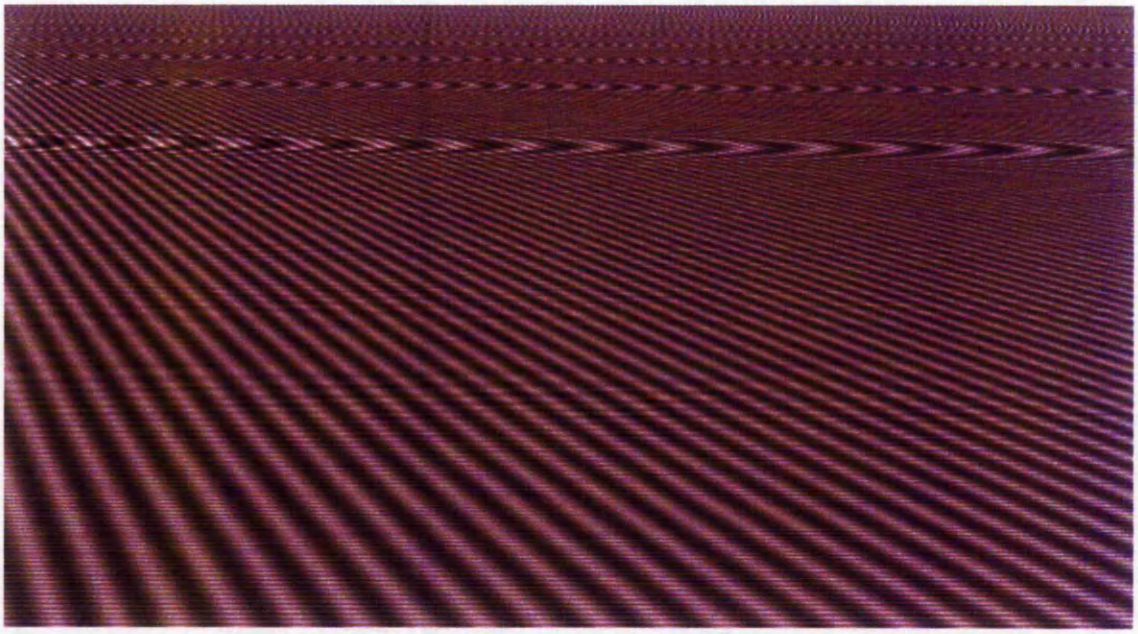


Figure 3.2 **(c) Aliasing effects are reduced by taking 16 samples at the same points within each pixel.**

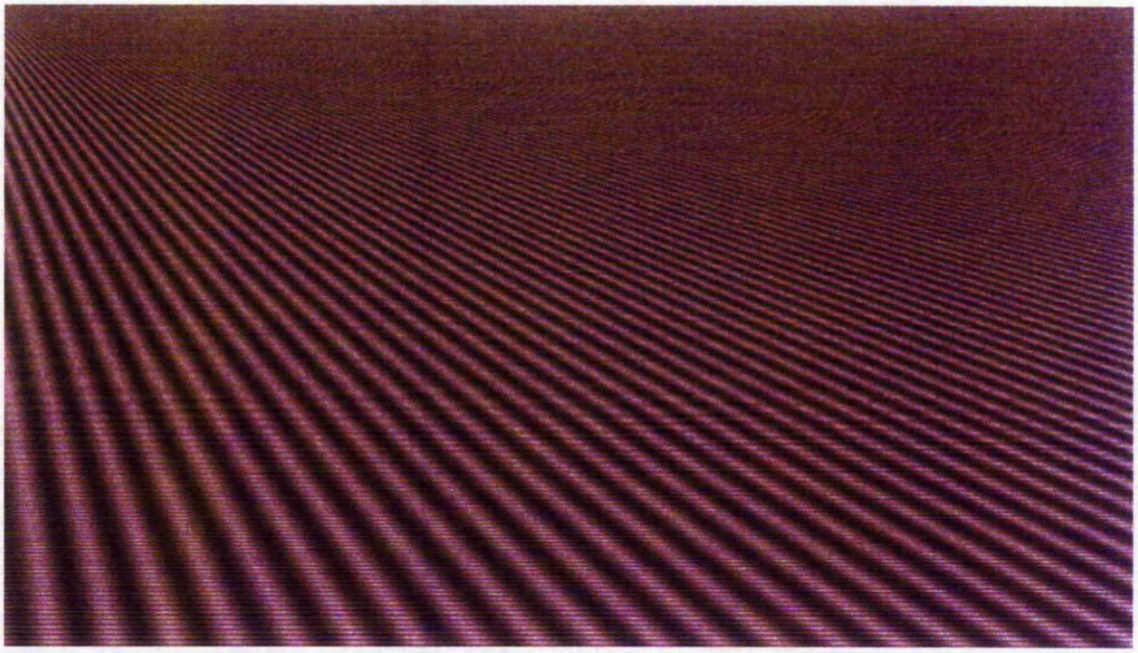


Figure 3.2 **(d) Effects of aliasing reduce further if the samples are taken randomly within each pixel.**

3.4 Established Methods of Anti-aliasing Texture In Real-time

3.4.1 MIP Mapping

Williams [WILL83] deals with the problem of texture aliasing by making use of several images of the texture at various resolutions, each of which are derived from the original by averaging down to lower resolutions. Each image in the sequence is at exactly half the resolution of previous in the linear dimension (i.e. a quarter of the number of samples of its parent). When a transformed screen pixel is covered by a collection of texture pixels (texels), the MIP map pixels within a table corresponding to this collection most closely are used to give a filtered colour value. This is obtained by bilinearly interpolating amongst the four nearest values within the chosen table.

Since only a limited number of the tables may be stored, values from two adjacent tables must be blended in order to deal with the difference of the transition of one resolution to another across a surface. This is achieved by accurately determining the level of compression within the MIP map and using this value to linearly interpolate between the two closest tables.

MIP mapping has the advantage of speed since only two bilinear interpolations are required to get a value from each adjacent table, plus an additional interpolation between the two values (resulting in trilinear interpolation). However, this is generally at the expense of accuracy. For example, a perspective projection may well require that the texture be compressed in only one direction. This will tend to result in obvious blurring of the original texture as the MIP map goes down to lower resolutions at greater viewing distances. Figure 3.3 illustrates the effect of blurring using a MIP map.

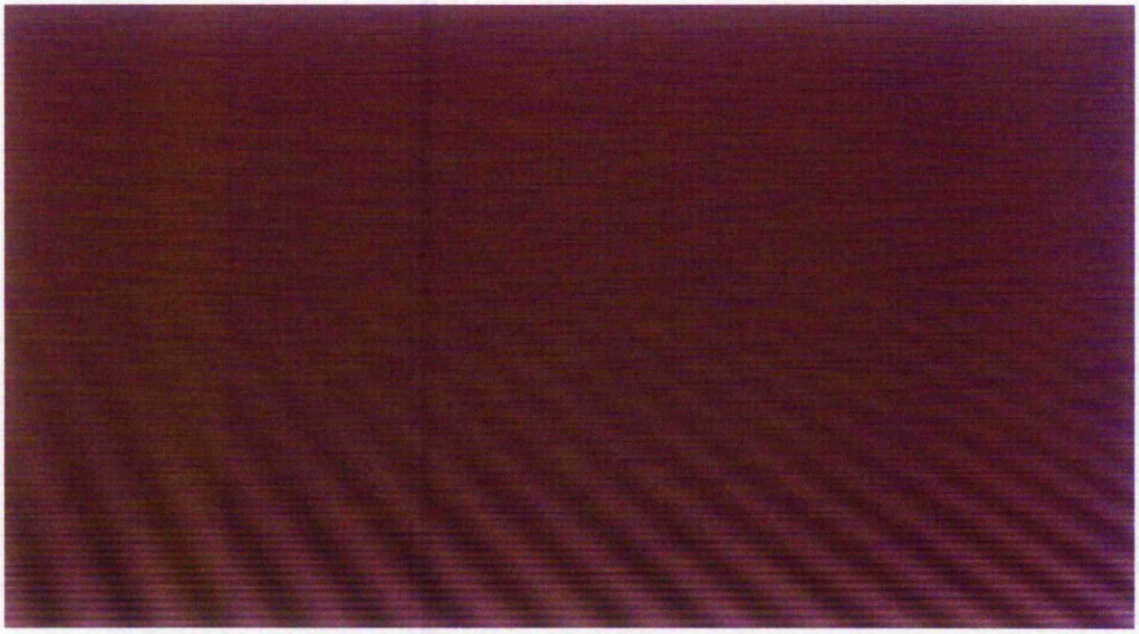


Figure 3.3 **Effects of aliasing are not easily observable here, but blurring occurs as the texture compresses with distance to the point where the texture becomes unobservable.**

3.4.2 Summed Area Table

Crow [CROW84] devised a scheme in which a single table of entries calculated from the original texture is used by which a sum of texture values lying within a given rectangle can be easily determined. Thus if we place a bounding rectangle around a transformed pixel in texture space, an average texture value can be determined by evaluating the sum of texture values within it using the summed area table and then dividing this value by its area. The idea of making use of such a table is neat in that it does not require us to actually look at each of the texture values within the rectangle at render time.

This method has an advantage over MIP mapping in that a virtually continuous range of texture densities can be obtained in two directions independently. This improvement is well illustrated in figure 3.5a, which has the same texture pattern and viewpoint as Figure 3.3, but uses the summed area technique in place of MIP mapping. The extra processing required to perform the Summed Area Table method comes from the fact that now four bilinear interpolations are required to map each of the corners of the screen pixel to texture space in order to obtain a bounding rectangle.

To see where the Summed Area method becomes significantly less accurate, consider a screen pixel that has undergone a perspective bilinear transformation to texture space. See figure 3.4.

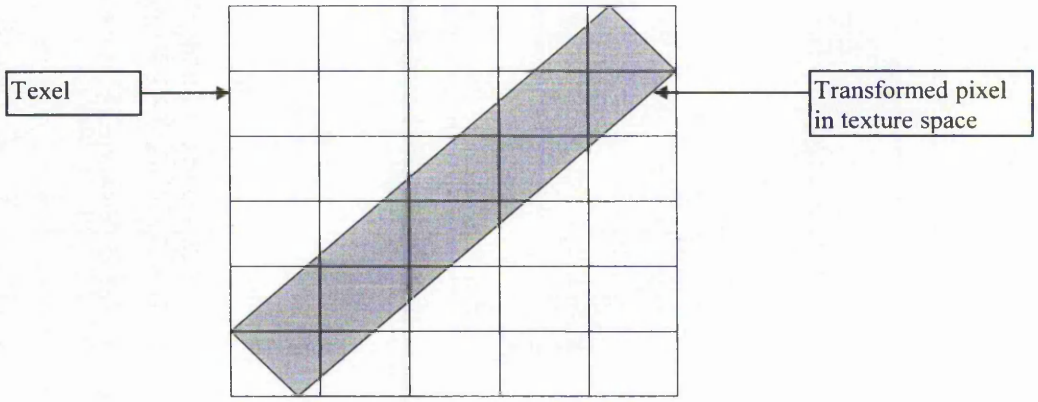


Figure 3.4 **Illustration of how the summed area table ignores substantial areas of the texture at certain orientations.**

It can be seen that a bounding rectangle will not suffice when the texture becomes compressed and the surface is viewed at rotations about more than one axis with reference to the surface. These cases therefore need to be taken into consideration since excessive blurring will be observed when they occur. This effect is illustrated in Figure 3.5b. The same texture pattern has been used as before, but this time it has been rotated by an angle of 45^{o} before being stored. This rotation has been reversed by the viewing transformation creating a long, thin pixel footprint at 45^{o} as shown in Figure 3.4 As with the MIP map, the outcome is that the vertical strips are suppresses sooner than is necessary.

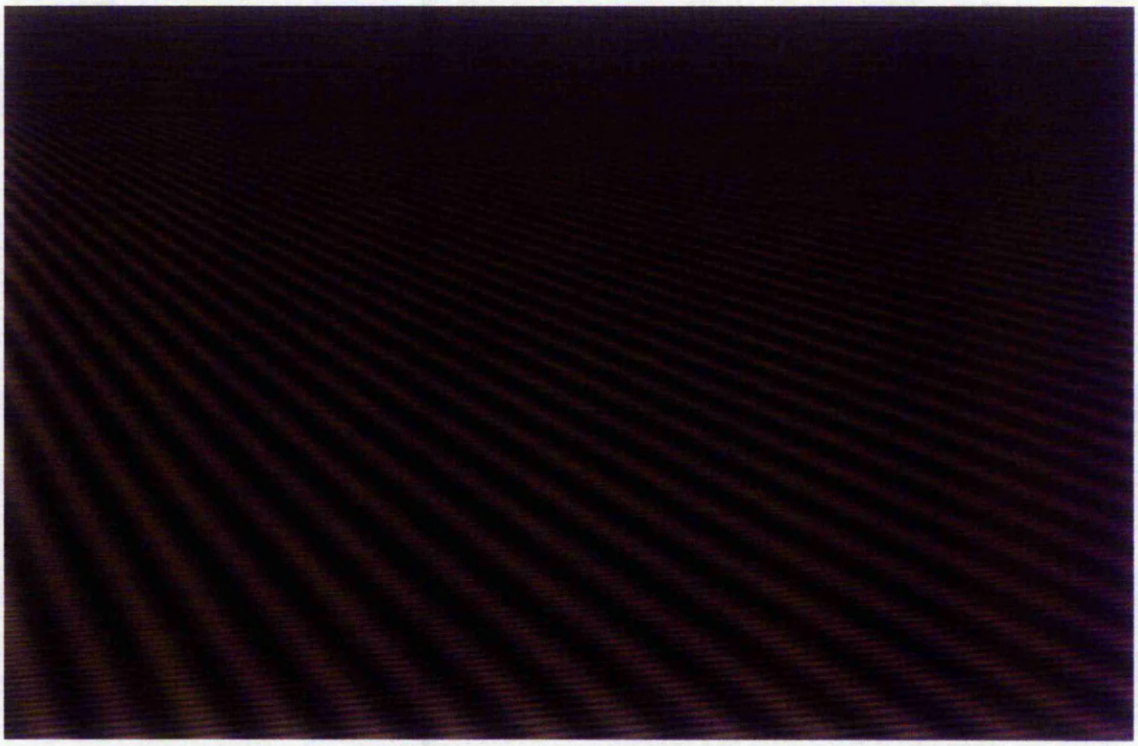


Figure 3.5a **Effects of blurring are removed with the summed area table - but this is a favourable orientation**

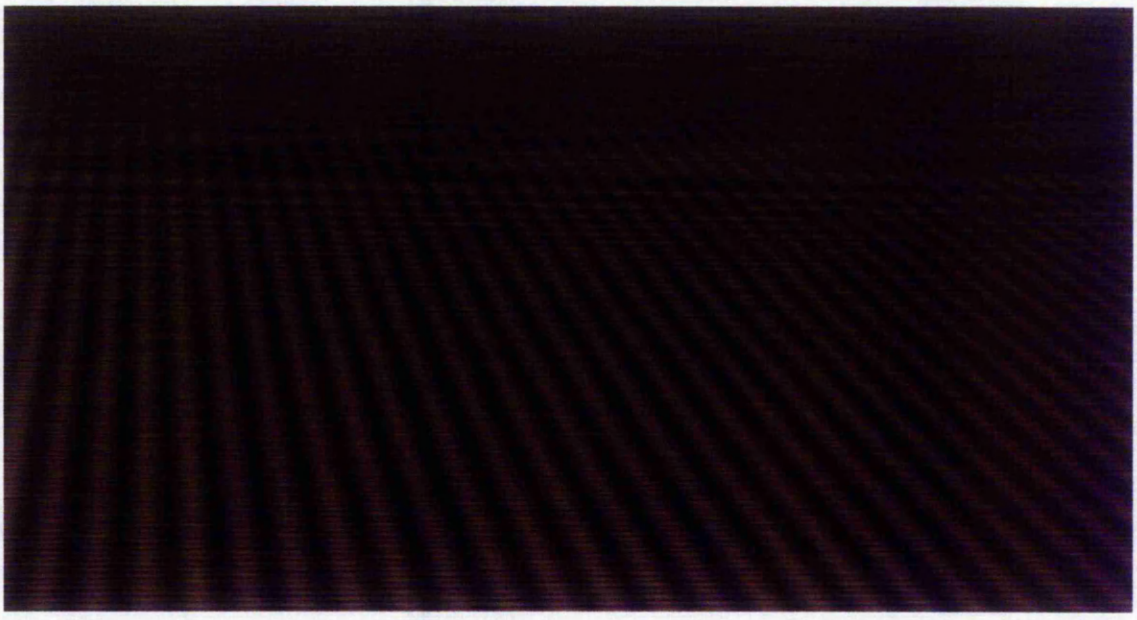


Figure 3.5b **Effects of blurring and fringing are evident here with the summed area method. This is due to the orientation of the texture and the type of texture used.**

3.4.3 Adaptive Precision Method

In an attempt to solve this problem, Glassner [GLAS86] refined the summed area method by iteratively trimming away the excess areas of the bounding rectangle. This requires some knowledge of the geometry of the transformed screen pixel, relative to its bounding box, to be detected. The excess areas are then subdivided into either triangles or rectangles that can then be deducted from the original sum (see Figure 3.6).

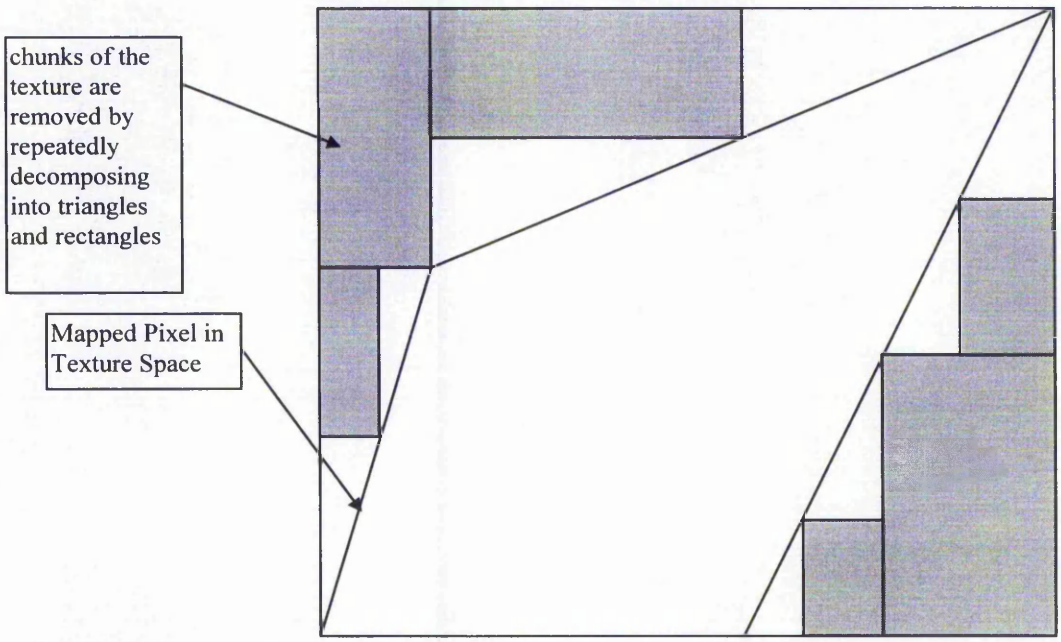


Figure 3.6 **Illustration shows how the adaptive precision method is used to reduce the texture to within the boundaries of the warped pixel.**

Although the method minimises the errors that result from the summed area method, it is found that producing an estimate on the number of subdivisions that need to be made for good results, is difficult to produce accurately. This will, therefore, lead to unpredictable results. Furthermore, the classification of general quadrilaterals tends to make the overall process algorithmically complicated. Such an algorithm would be difficult to optimise or implement in hardware. A particular problem also arises from the large number of accesses, which may need to be made to the table for each pixel. In a highly optimised system, these accesses will be the dominant factor in determining system performance.

3.4.4 Footprint Assembly Mapping

The assembly mapping technique [SCHI96] approximates a circular pixel in screen space to a parallelogram in texture space. This parallelogram (or footprint) is based on an inscribed ellipse that is the result of a texture transformation of the circle without perspective deformation. Figure 3.7 shows how the parallelogram is formed based on the transformed pixel.

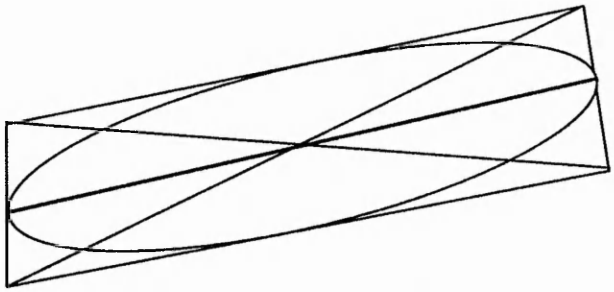


Figure 3.7 **The transformed pixel footprint is approximated as a parallelogram.**

Since the dimensions of the parallelogram are based on the major and minor axes of the ellipsoid, they are too difficult to calculate accurately in real-time. Therefore, the dimensions and orientation of the footprint are approximated using the largest value of the rates of change of u and v with respect to screen pixel co-ordinates.

Once the footprint has been approximated, the transformed pixel is approximated still further by splitting it up into a number of texel squares in which MIP map samples may be taken (whose centres lie on the major axis of the parallelogram). The number of these square MIP mapped texture elements for the footprint is based on the relative proportions of the dimensions of the parallelogram so that:

$$\textbf{No. of MIP map square samples} = \frac{\textbf{major axis of ellipse}}{\textbf{minor axis of ellipse}}$$

Figure 3.8 shows how the MIP mapped texture elements are formed based on the shape of the footprint. We see that long, thin transformed pixels require many more samples

compared with more rhombus shaped footprints. Note also that the number of MIP mapped texture elements are rounded to the nearest power of 2 for reasons of practicality.

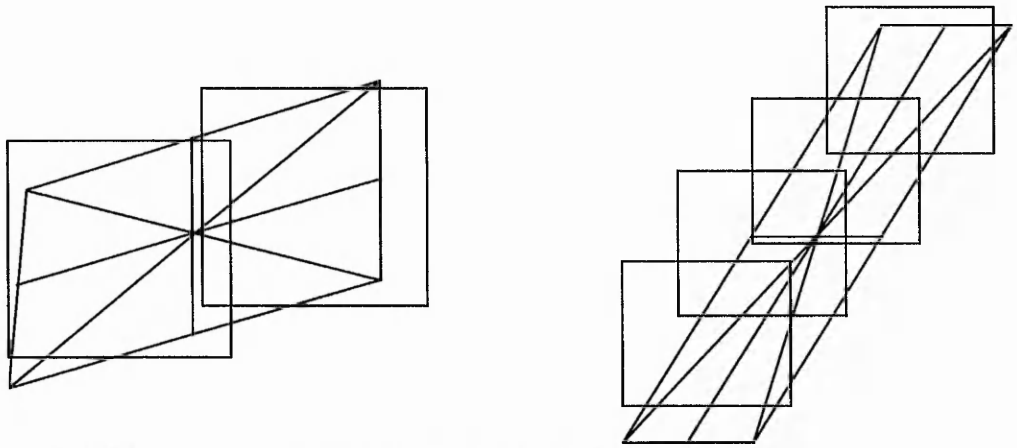


Figure 3.8 **The relative proportions of the left footprint require fewer MIP map samples compared with the right footprint. The square boxes represent MIP samples.**

This technique can provide good quality results and has been implemented in hardware. However, at certain viewing orientations of the texture, a great many samples need to be taken. As a result, the number of samples cannot be allowed to get too large if real time performance is required and hence the quality must be reduced.

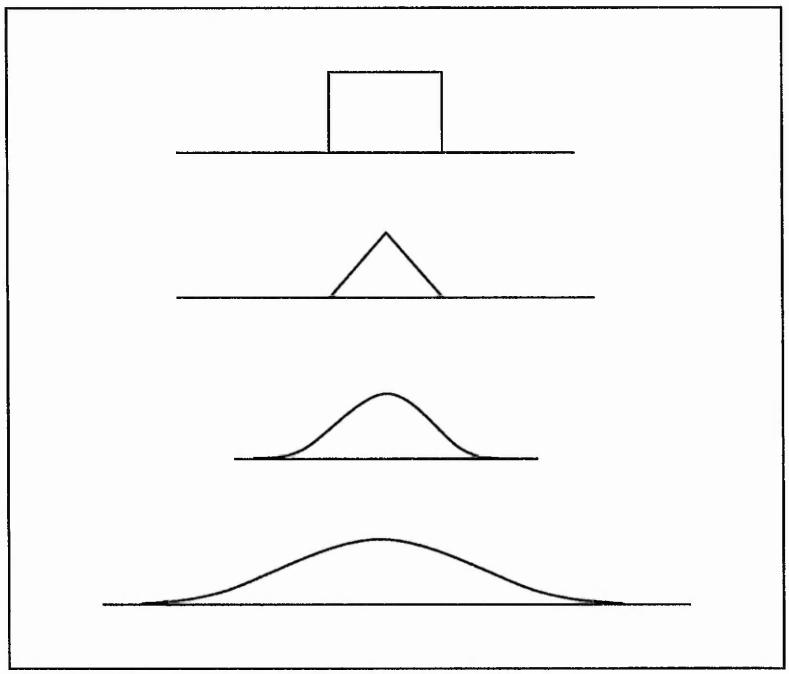
3.4.5 Other Methods of Anti-aliasing Texture

Other well established methods have less application in real-time systems since they tend to work in texture space (such as Feibush-Levoy-Cook [FEIB80]). This means that a large number of transformations from texture space to screen space may well occur for a single pixel, in which the cost per screen pixel is proportional to the number of texture pixels accessed. For the sake of completeness, some of these methods are discussed briefly, since they have some application with regard to the analysis of texture aliasing.

All methods of texture anti-aliasing make some use of filtering to resample the image. This is because filtering addresses the causes of aliasing rather than its symptoms (which point sampling at a higher resolution does not). The cross sectional shape of the filter used determines the quality of ant-aliasing and theoretically, the ideal low pass filter $\text{sinc}(x) = \sin(\pi x)/ \pi x$, but its infinite width makes it impractical for computation. In practice, a finite impulse response filter must be used. The most commonly used of this class of filters are

the box, the triangle, the cubic B-spline and the truncated Gaussian. Figure 3.9 shows the shape of these filters respectively.

Figure 3.9 **Cross sections of some of the common texture filters, ordered by quality.**



The filter function used by Feibush, Levoy and Cook [FEIB80] saves time on calculation by making use of a pre-calculated table of lookup values. This means that a more elaborate filter function can be employed, such as a Gaussian form. The filtering algorithm uses the following steps at each screen pixel:

- Centre the filter function on the screen pixel and find its bounding rectangle
- Transform the rectangle to texture space, where it becomes an irregular quadrilateral and find its bounding box.
- Map all texels inside the rectangle to screen space
- Form a weighted average of the mapped texels using a lookup table, which is indexed by each samples location within the pixel.

In order to reduce the number of texture accesses that need to be made, Greene and Heckbert [GREE86] mapped an "elliptical weighted average" filter into texture space. The

filter shape is a circularly symmetric function in screen space and is warped by an elliptic paraboloid function into an ellipse in texture space. Once the elliptic paraboloid is in texture space, incremental methods can be used and thus, only a few arithmetic operations need be used per texel.

3.4.6 Summary of Texture Mapping Techniques

The algorithms presented in the previous sections are now summarised in table 3.1 in order to gain an overview of the status of texture mapping techniques.

Technique	Advantages	Disadvantages
Point to point sampling	Fast, easy to implement	Aliasing and scintillation effects
MIP mapping	Fast box filtering	Over-estimates of footprint size leads to over-blur.
Summed Area Table	Fast and efficient	Table sizes are very large with blurring still evident
Adaptive Precision	Reduces effect of over-blur	Algorithmically complex
FootPrint Assembly	Good quality at fair performance	Large sample counts force reductions in quality.
Convolution methods	Very high quality	Not realistic in real-time

Table 3.1 **Comparison of advantages and disadvantages of most popular mapping algorithms.**

3.5 New Method: Potential Mapping

Having examined the range of texture anti-aliasing methods, it becomes evident that there is a bias either in favour of efficiency, leading to spurious areas of the resulting texture, or in favour of texture integrity, where computation is greatly increased and hardware implementation becomes impractical. It would therefore seem appropriate to balance the scales of efficiency and integrity by developing a new method that takes all of the following factors into account:

- **A simple algorithm.**
- **Ability to cope with general texture patterns at any orientation.**
- **Speed.**
- **Ability to implement in hardware.**
- **Minimal Aliasing**
- **Minimal Blurring**

Potential mapping is based on the idea of finding a quick way of performing the integral of the texture pattern over the area of the transformed pixel, which avoids the problems of the summed area table without introducing too much algorithmic complexity.
If we assume a screen pixel to be square then its resultant image after rotation and a perspective transformation of its corner points will be a general quadrilateral. Ideally we would like a method that deals with the quadrilateral directly without approximating it to any simpler shape. Furthermore, as with Crow, we would like to avoid having to look at each and every texture pixel that lies within the transformed pixel when determining the average texture value (i.e. when finding the texture sum and the area of the quadrilateral).

Consider Gauss' Law in used physics. This states that the total outward electric flux over any closed surface is equal to the free charge enclosed by that surface. In other words, the charge contained within a volume can be found by traversing *only* the surface of that volume. The law can be written symbolically as:

$$\oint \vec{D}.\vec{ds} = Q$$

This is an extremely useful relation in the field of electromagnetism and surprisingly, we can use the principles of Gauss's Law to derive a new method of texture anti-aliasing. If we reduce the integral by one dimension then, by analogy, the texture contained within a transformed pixel can be found by traversing only its boundary. This is essentially the principle behind Potential mapping.

Potential mapping for a given screen pixel is done by tracing around the boundary of its transformed pixel in texture space. Whilst tracing the pixel from left to right, we force each texture co-ordinate to have a corresponding "potential" value which is based on the actual texture intensities down a column in texture-space (see figure 3.10). The total texture intensity bounded by the transformed pixel within that column is then found from the corresponding upper and lower "potential" values. If we integrate this over all the columns within the transformed pixel, an average intensity value can be deduced. Since the "potential" values only need to be calculated once for each texture map, they can be pre-computed and stored, leaving only one operation per texture column and only one division per transformed screen pixel in order to average over the area.

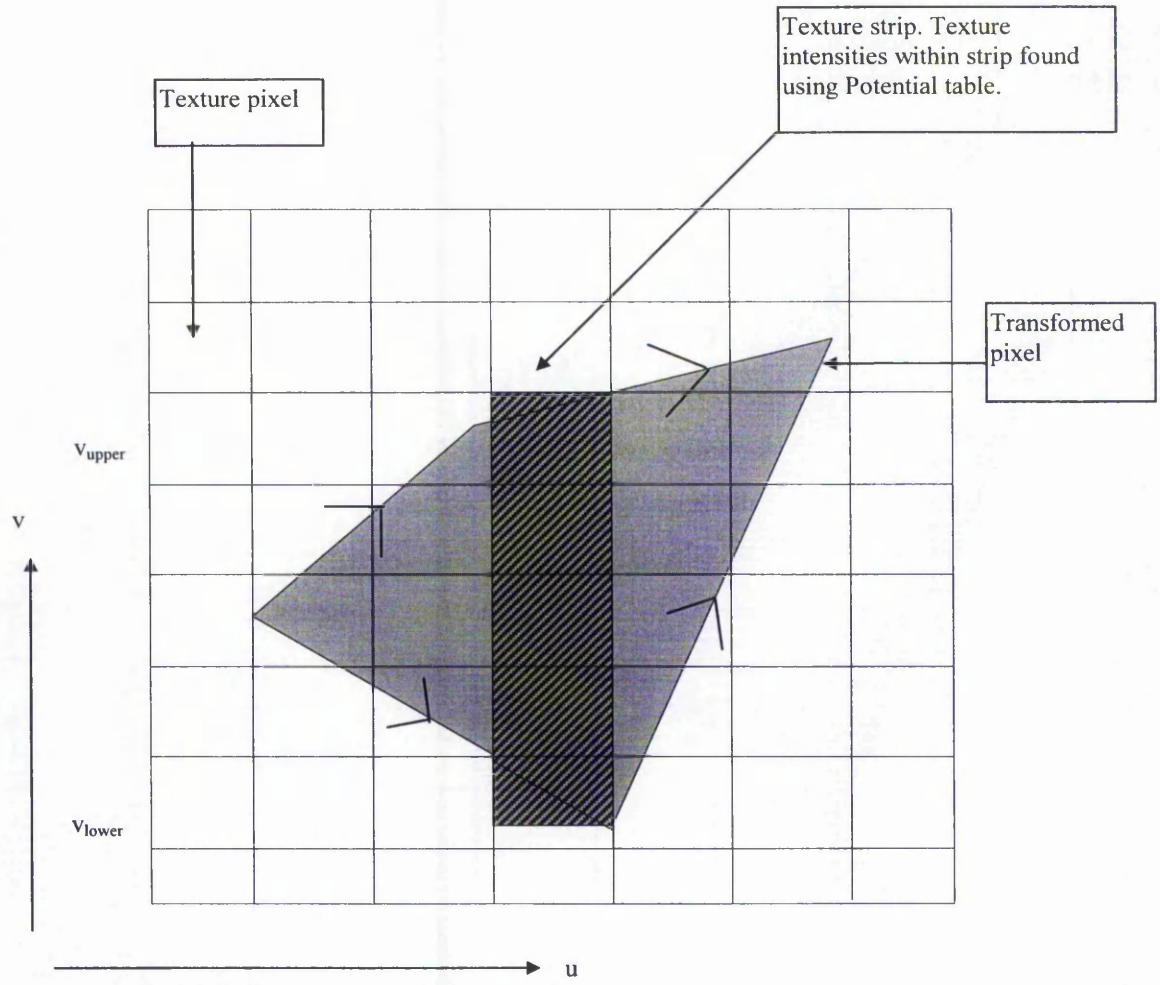


Figure 3.10 **Illustration showing how texture strips are found whilst the texture pixel is traced (note arrowheads around pixel boundary).**

The method can be summarised algorithmically:

for u (left of transformed pixel to right)
find vlower and vupper
find "potential" across texture column, P(vlower, vupper)
Texture Sum = Texture Sum + P(vlower, vupper)
Area = Area + (vupper - vlower)

Average Intensity = Texture Sum / Area

The algorithm clearly shows that the problem of finding the summed area has now been reduced to one dimension (i.e. the u direction) such that the number of iterations required becomes solely dependent on the width of the transformed pixel. Note that we could alternatively make use of horizontal texture strips and sum in the v direction just as easily.

The task of implementing Potential mapping optimally can essentially be divided into two: the construction of a table of 'potential' values such that $P(v_{lower},v_{upper})$ is computationally inexpensive, and finding v_{lower} and v_{upper} as we trace around the pixel.

3.5.1 Evaluating Potential Values

As the boundary of the transformed pixel is being traced in texture space, the summed texture values lying within each vertical strip need to be evaluated. This can be achieved by building up a table of "potential" values. These values are derived from the texture by storing a cumulative sum of texture values down each column of the original texture. Figure 3.11 illustrates this:

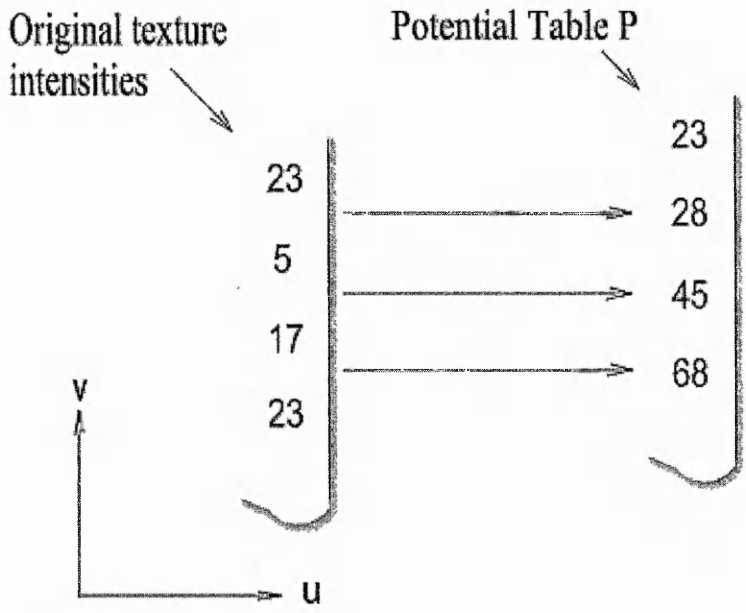


Figure 3.11 **Illustration shows how the potential table is formed for later use.**

Now we can find the 'potential difference' (or summed texture), P, across the upper and lower texture values in a vertical strip at horizontal texture position u (see figure 3.10) by simply subtracting the upper and lower potential values corresponding to v_{upper} and v_{lower}:

$$\mathbf{P(v_{lower}, v_{upper}) = p[u, v_{lower}] - p[u, v_{upper}]}$$

Where the values of **p** are pre-stored in a potential table.

3.5.2 Tracing around Transformed Pixels

As with Crow's method, the four corners of the screen pixel need to be mapped into texture space (Note: we find that only one corner need be mapped for pixels lying within a screen polygon, see later). One way of performing the tracing operation is to start with the extreme top left corner of the transformed pixel in texture space. We then find the gradients to its nearest lower and upper corners and next the gradients from those corners to its farthest right corner. Since we are ultimately dealing with discrete texture pixels, the values of v_{lower} and v_{upper} can be evaluated trivially using unit increments in u. This method is better known as the DDA algorithm and has performance drawbacks since four divides must take place in order to find each gradient, with the process of rounding v to an integer also consuming time [FOLE90].

Bresenham Tracing

The need for divide operations can be removed by using Bresenham's line drawing algorithm [BRES65]. This algorithm avoids floating point calculations during tracing with the advantage of guaranteeing accuracy by continually minimizing the error between the traced edge and the true line (thus avoiding double-counting of contributions).

When the slope of a line is greater than 1, the usual procedure is to exchange the x and y (in this case u and v) co-ordinates and plot the line "on its side". In the present case, we do not wish to do this because it would result in more than one entry per column. To compensate for this, a complicated calculation of the integration measure would be needed. Instead, we just continue to use the algorithm in its original form, allowing near vertical lines to be sparsely populated.

The consolidation of subpixel displacements into whole pixels requires a division for each step along the line. In the normal Bresenham algorithm [BRES65] this is reduced to a simple subtraction because we know that the only possible answers are 0 and 1 for a line at less than 45 degrees. In the present case, we do not have this restriction but we can still know the answer in advance to an accuracy of 1 unit if we calculate the integer part of slope first. This might seem to cancel the original advantage but because the gradients are nearly the same for all pixels in a scan-line in screen-space, this can be done just once per line. Similarly, left and right gradients for pixels in a column are the same can be shared(see figure 3.12).

We can make further efficiency gains by noting that adjacent pixel corners are shared between screen pixels, so that only one bilinear mapping need be made for each pixel except at the edges of the polygon. This commonality also applies to the pixel edges, allowing a possible saving of a factor of two in the time taken to trace around each pixel.

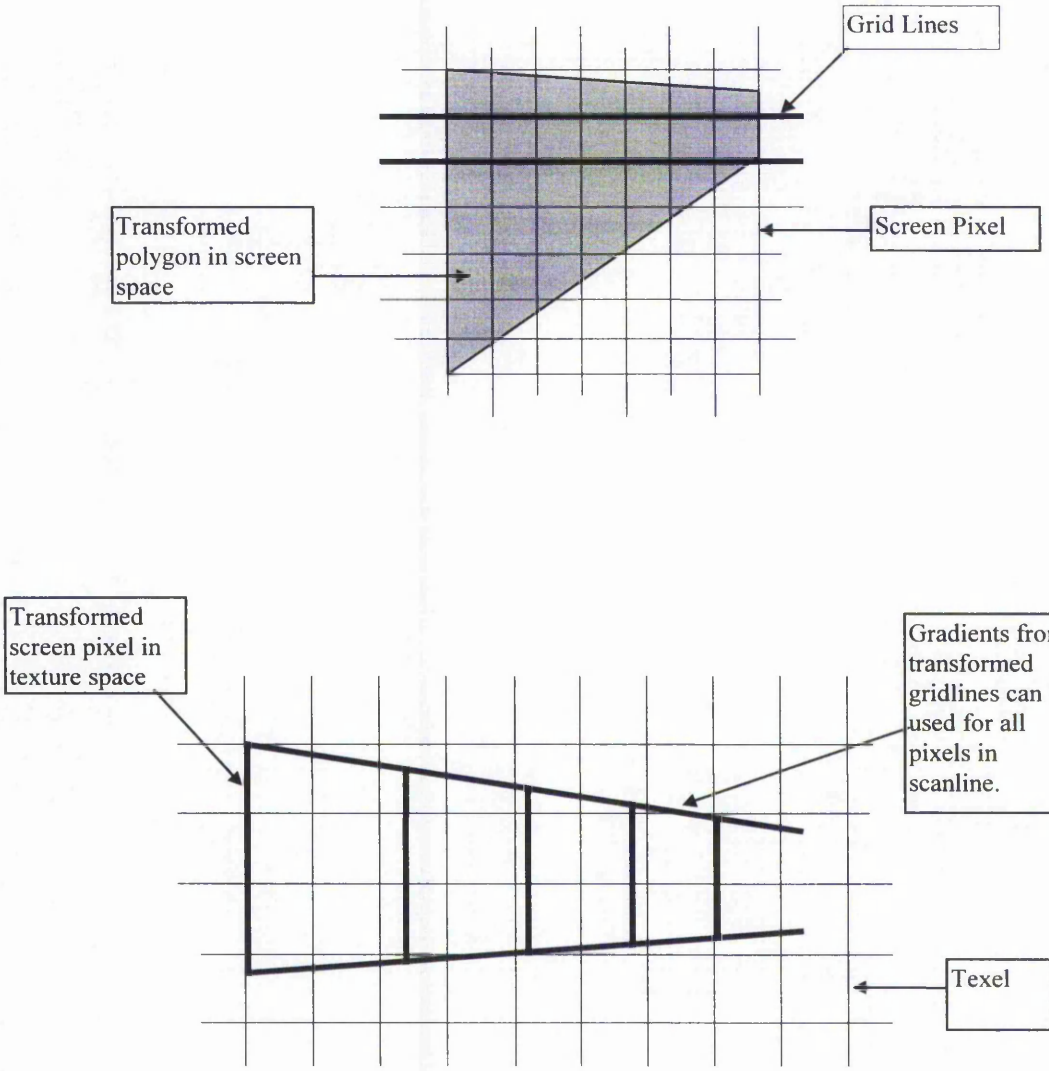


Figure 3.12 **Illustration showing how the cost of finding the gradients around each transformed pixel in texture space can be reduced by mapping gridlines in screen space prior to texture mapping.**

There is one particular case when the calculation leads to a summed texture of 0! This occurs when the fractional values for v_{lower} and v_{upper} lie within the same texel. In order to avoid modifying the algorithm to overcome this problem (since we would like to keep it as simple as possible for hardware purposes), we can use the fact that the fractional v_{lower} and v_{upper} values are always distinct when texture mapping takes place. If we double the resolution of the potential table in the v direction so that the new table values are the average of consecutive pairs of the old table values, then we can simply force upper values of v in each strip to access the upper values in the table, and the lower values of v to access the lower table values. Thus, we ensure that the summed texture is always greater than

zero without incurring any modifications to the algorithm. To continue to exploit the commonality of edges between neighbouring pixels, we must arrange for the two sets of values to be retrieved during the same memory access. Since potential maps are generated off-line, we simply change the format to ensure these values neighbour each other when they are saved. Note that when tracing around pixels that lie on the *edges* of a polygon in screen-space, it is possible to use the polygon edge rather than the pixel edge. We can use therefore texture map the geometry data of pixel fragments in the manner chapter 2 in order to provide improved anti-aliasing of the edges of textured polygons. This is a further advantage of Potential mapping compared to the MIP map since it pre-filters the texture and cannot "know" where the polygon boundary will lie.

Results of the potential mapping algorithm show that the effects of blurring and aliasing are significantly reduced when compared with MIP mapping (figures 3.3 and 3.13a respectively) and when compared with the summed area method (figures 3.5 and 3.13b respectively). The residual Moiré fringing which can be seen in these figures is the result a top-hat style filter which has been used. To eliminate this fringing it would be necessary to use a Gaussian form filter, which has a more gradual fall off at the edges. Unfortunately, this is difficult to achieve without resorting to the brute force convolution techniques described in chapter 2. In this case, all the texels that lie within and around the projected pixel edges would need to be considered.

Since the texture patterns used in these images represent a severe test of a texture anti-aliasing system it is likely that the kind of patterns used more commonly in practical applications will not display the effect so strongly. If the residual fringing *is* a problem, it can easily be eliminated (at a cost of slight blurring) by passing the image through a filter before display (in practice, an adjustment of the monitor focus can achieve the required effect at zero cost!).

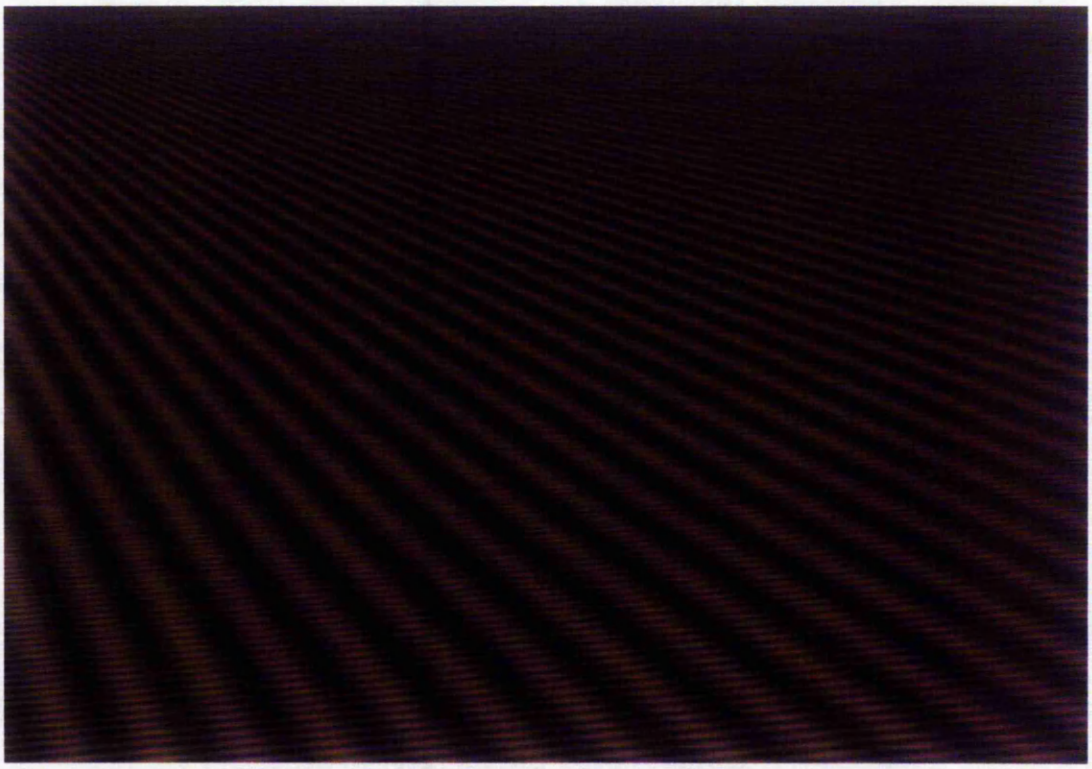


Figure 3.13 **(a) Using the Potential Mapping Technique the excessive blurring noticed in figure 3.3 is removed without introducing significant aliasing (the image of the source texture is provided in figure 3.2)**

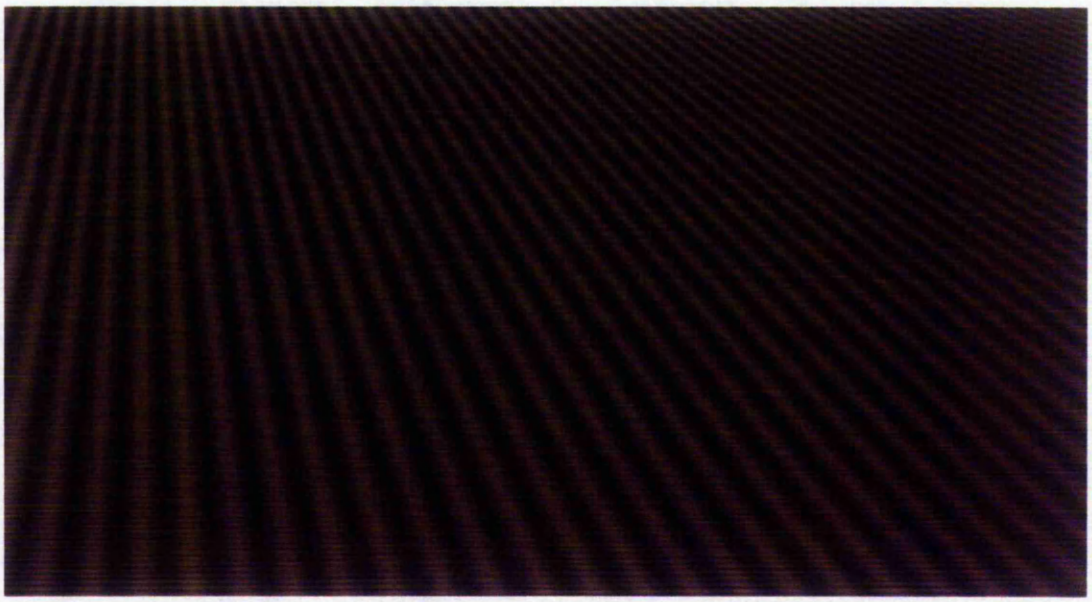


Figure 3.13 **(b) The effects of blur, aliasing and fringing are greatly reduced when using potential mapping, even with the unfavourable texture orientation, as used in Figure 3.5b (the image of the source texture is provided in figure 3.2).**

3.5.3 Implementation and Performance Considerations

Test programs written for TPM research also include extra instructions to count the amount of time taken by each stage in the process (note that section 3.7 provides an in-depth comparison between competing algorithms). For a full screen image of texture at a shallow angle (as shown in the illustrations above) the results show that the inverse perspective transformation takes 12% of the time, set up operations take 40% and the remaining 48 % is spent tracing around the edge of the pixels. At a less optimal orientation of the texture map, (i.e. when the projected pixel is very long in the non-integrated direction) the time spent tracing increases by 50%. If speed is more critical than memory size, then this problem can be overcome by creating an alternative texture potential that is summed in the opposite direction. Since the time taken to perform the inverse perspective transformation will be the same as for the other algorithms, and the set up operations that need to be done for each pixel are likely to be comparable to those required by the summed area table method, these proportions give an approximate guide to relative performance. This suggests a degradation of a factor of two in speed compared to the summed area table and between two and four compared to MIP mapping, depending on how many samples and scales the MIP map uses. When the texture system is being used optimally, the texels will be of a similar scale to the screen pixels. In these circumstances, the number of entries that need to be summed for each pixel edge will normally be only one or two. Consequently, the time taken to trace around the pixels will disappear and TPM will be only marginally slower than the summed area table or MIP Map.

All these estimates assume that all instructions other than divide and reciprocal take a single cycle and no allowance has been made for cache misses. In many systems, the external memory is much slower than the cache and instructions, which do not access it, run very much faster than those that do. In such a system, the number of external memory accesses provides an alternative performance measure. It is likely that the only external accesses in any of the algorithms will be the reads from the texture map or potential.

On a single processor using current technology, none of the high-quality algorithms are capable of real-time performance. Therefore, interactive systems will require multiple processors or special hardware. TPM is well suited to parallel processing using a pipeline

architecture because it easily decomposes into the stages of transformation, pixel edge set up, pixel edge tracing and the summation of the final results.

A subdivision of texture data can also be achieved by giving multiple texture processors a separate region of the texture map. This would be easy to implement from a processor networking perspective since the only data to be communicated between texture processors are the sums over pixel edges and they only need to be between nearest neighbours. Multiple copies of the potential map would be needed to avoid access conflicts between the processors. Strategies for hardware implementations of texture potential mapping will be revisited in more detail in chapter 5, which focuses specifically on parallel schemes.

3.5.4 Implications of Texture Potential Mapping

As remarked above, the justification for using the potential mapping method is that it is be faster than existing high-quality methods whilst producing better quality results than existing high-speed methods.

Whilst the new method is somewhat slower than those implemented current mainstream hardware, it is interesting to note that this speed differential is outweighed by the technological advances of recent years. Consequently a potential mapping system implemented with current technology would have a better price/performance ratio than would have been possible using MIP mapping or the summed area table at the time at which they were first proposed.

The main criticism of TPM as it has been presented so far, is that a dependency exists between the orientation of warped pixels and number of texture samples required to perform integration. This starts to become an issue at more extreme orientations. The current implementation does not facilitate any means to control the quality and performance required. The following sections of this chapter refine TPM.

3.6 Extensions to Texture Potential Mapping

A drawback of the texture potential mapping algorithm is that the number of samples taken in the u direction of a transformed pixel is dependent on the width of that pixel. This means

A drawback of the texture potential mapping algorithm is that the number of samples taken in the u direction of a transformed pixel is dependent on the width of that pixel. This means that, at certain viewing orientations, a great many samples may be required for long, thin pixels (although the time taken is still less than a brute force method). When these pixels are extremely distant relative to the viewer, we would like to able to limit the number of samples taken in order to improve performance without incurring a degradation of image quality. Furthermore, we would like to allow the user to control the best compromise between quality and speed [CANT2000].

3.6.1 Texture Potential Mip mapping

The TPM algorithm produces good qualitative results, as shown on the left hand side of Figure 3.14b using the source texture shown in figure 3.14a. This is more evident when we compare with trilinear MIP mapping of an identical image shown on the right hand side of the figure. However, one criticism that could be made of TPM, unlike methods such as MIP mapping or the summed area table, is that the time taken per pixel by texture potential mapping will vary according to the scale and orientation of the texture. This is due to that fact that integration occurs in one direction only leading to an unpredictable variation in the number of samples that need to be taken in the other direction. Texture Potential Mapping thus has a weakness in that, when the projected pixels are very wide the number of samples required, although smaller than needed by the brute force method, is still excessive. In many of these cases, the summed area table, or even conventional MIP mapping can produce quite adequate results.

Figure 3.14a **The source texture used for subsequent qualitative analysis of texture mapping techniques. It should be noted that a qualitative match to the electronic versions of the images would be slightly affected by the printing process.**

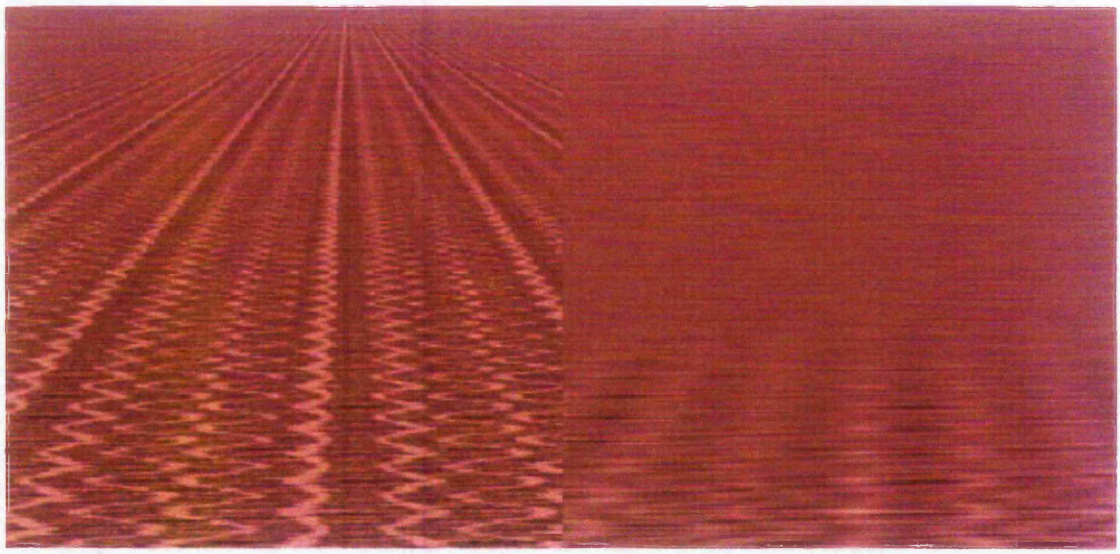


Figure 3.14b **Texture Potential Mapping of a pattern at an extremely shallow angle compared to trilinear Mip mapping of the same pattern at the same angle.**

3.6.2 Combining TPM and Mip mapping

To overcome this problem, the combination of texture potential mapping with a MIP map approach in the non-summed direction is presented. The algorithm has been named Texture Potential MIP mapping (TPMM). The pre-processing phase produces a set of potential maps, each one half the size of the preceding one in the x direction only. The summation principle is used in the y direction. The resulting table is twice the size of the original potential map. The generation of this table from the original texture pattern can be done in either order but it is more efficient to filter and decimate for the MIP map before constructing the potential since it means that the filtering process can be done with lower resolution numbers.

Rather than box filtering in the image domain, the filtering has been done by Fourier transforming the original texture, deleting frequency components above the Nyquist frequency for each MIP map and then transforming back to the x space representation. This method of filtering is time consuming but has the advantage of guaranteeing optimal results. Similar results may be achieved with an FIR (fast impulse response) filter. An FIR filter, is a digital filter for which each output sample is a weighted sum of a finite set of input samples. The array of weights (known as coefficients or taps) have the same form as the impulse response of the filter. Such filters work by convolution, but can perform any

transformation of the amplitude or phase spectrum (in our case, we we would require a low pass filter). This method, however, would require a convolution window as large as the texture pattern to achieve the same results.

To generate an anti-aliased, textured pixel we first apply the texture transformation to the corners of the pixel. In an optimised implementation, this would be done incrementally and the results from corners that are shared between pixels can be re-used. The workload for this operation is thus only marginally greater than it would be for a non-anti-aliased texture implementation. In fact, this extra amount relates precisely to the margins of pixels!

Next, the position of the extremes of the pixel in the x direction of texture space is determined. This information will assist later in the process of edge tracing but its immediate purpose is to determine the width of the pixel. From the logarithm of the width of the pixel to the base 2 we can determine which of the MIP maps to use. The logarithm need not be computed exactly and so all we need to do is to find the most significant bit that is set. This would be easy in a hardware implementation whilst in software, if the width of the pixel was expressed in floating point format, the exponent would contain the information that we require.

At this point the procedure varies from that employed by normal MIP mapping, because in the present case there is no unique "correct" choice of table to use. The decision must be based on a trade-off between the accuracy with which we follow the edge of the pixel and the number of samples taken. The intuitive answer to this would probably be to choose the table so that the number of pairs of samples taken lies between, say, 4 and 8 or 8 and 16. This can be done using a parameter $\boldsymbol{n}$, with the maximum number of sample pairs being limited to 2^n. Clearly if we were to allow fewer than four pairs of samples there would not be much chance of following an angled pixel footprint with any accuracy, whilst a number much greater than sixteen implies an excessive amount of work.

3.6.3 Aliasing Problem - Solution

In order to choose an acceptable balance between quality and performance, we need to understand how the image will be degraded if too few samples are chosen. The possible observable effects that could arise are either aliasing or blurring of the texture. Clearly, it is more acceptable to have blurring than aliasing but unfortunately a naive implementation of TPMM does in fact give rise to aliasing in the middle distance at critical angles as shown in Figure 3.15.

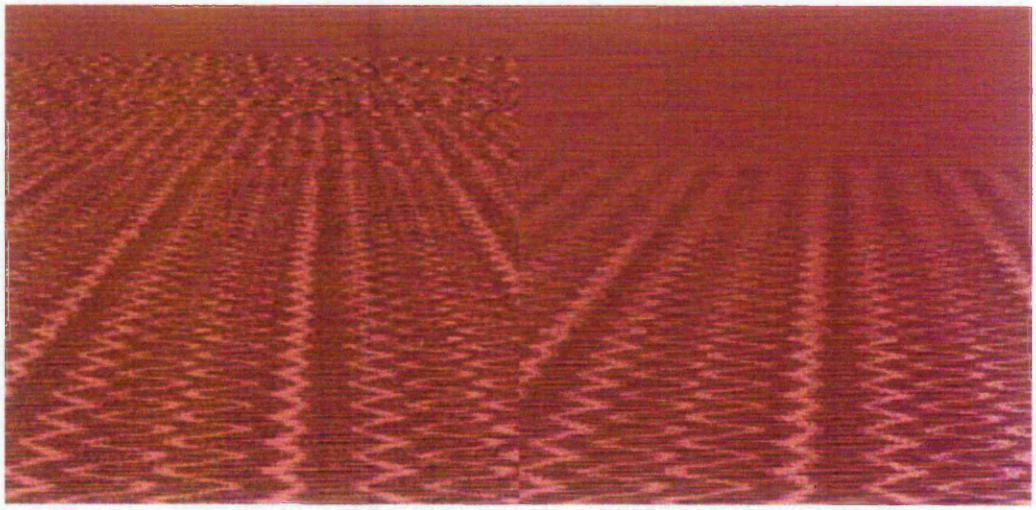


Figure 3.15 **Illustration of the aliasing effect in a naive implementation of Texture Potential MIP Mapping (left hand image) compared to the more acceptable blurring effect (right hand image), which replaces it when the traced footprint is enlarged to guarantee the inclusion of the whole of the pixel. Both of these images were created using a very low sample count to illustrate the effect clearly.**

To understand why this is so, consider the way in which the tracing process is affected by going to one of the coarser maps and taking just a few samples as shown in Figure 3.16.

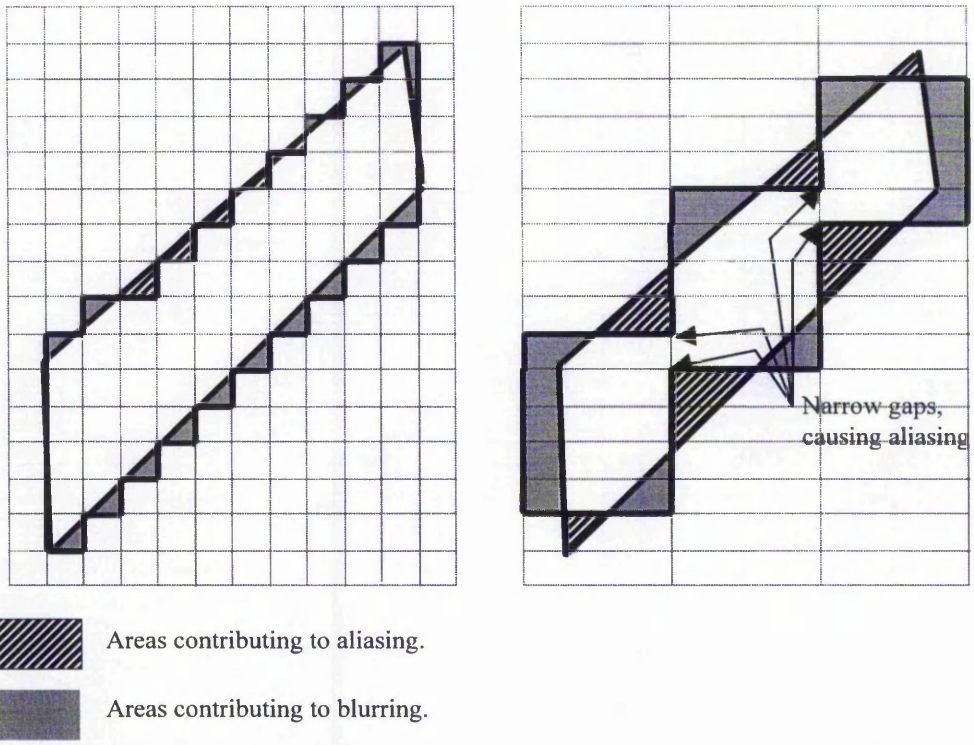


Figure 3.16 **Aliasing effects in the naïve implementation of texture potential MIP mapping. Left: Anti-aliasing at the resolution of the potential map. Right: lower map resolutions used with a MIP map.**

Where the traced pixel lies outside the ideal pixel, the result will be blurred because the sampling footprint is too large and hence the frequency cut-off too low. These areas are shown shaded on the figure 3.16 (left and right). The hashed areas show where the ideal pixel outline is outside the traced pixel. This is much more significant because in this case the footprint is too small and hence the frequency cut-off too high, potentially allowing in frequencies above the Nyquist frequency and causing aliasing.

The left hand side of figure 3.16 shows what happens if the original (basic) resolution of the potential map is used. In this case, the errors are very small and the resulting image will show neither aliasing nor blurring. The right hand side of figure 3.16 shows the situation when the texture value for the same pixel is calculated using one of the coarser MIP maps. The errors are much more significant now and the size or the hashed areas (which cause aliasing) is significant.

To correct this situation, the traced pixel must be *expanded* to guarantee that it includes the entire ideal pixel. This arrangement is shown in Figure 3.17. The total sample footprint is increased in the vertical direction. When this is done, the aliasing that was evident in Figure 3.15 is replaced by blurring.

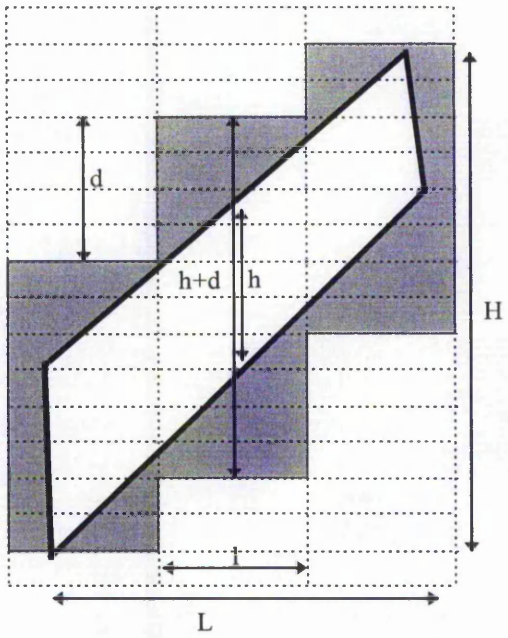


Figure 3.17 **Expanding the sampled pixel to cover a warped pixel.**

The coordinates that are used to access the table can be derived by any line drawing technique following the edges of the pixel. As with pure TPM presented earlier, either the DDA algorithm or Bresenham's algorithm can be used, depending on available hardware and bearing in mind that the slope calculation can be shared along a raster line. To expand the footprint to include the whole of the ideal footprint we calculate the positions of the intercepts at both edges of each column in the potential map and choose the largest value at the top and the smallest value at the bottom. Note that these intercept calculations are shared between neighbouring columns of texels.

At this point, it would seem that only one task remains - to find the optimal number of columns, n for a given application. This is not the best way to proceed however because there is an observed relationship between the ideal value of n for a given image and the angle which the pixels' long axis makes to the grid of the texture map. Very steep and very shallow angles allow a much smaller value of n to be used while near 45 degree angles

require a larger value. The reason for this is that in the former situations the ratio between the area of the ideal projected pixel footprint and that of the expanded footprint, is close to 1. A fatter pixel footprint also allows n to be smaller because the errors, although they have the same absolute values, are reduced as a proportion of the total pixel area. Figure 3.18 illustrates this point.

There are thus many situations in which a smaller value of n can be chosen than that which is required to cope with the most awkward cases. This is not surprising since these are the configurations where Crow's method or (in the fat pixel case only) traditional MIP mapping also work quite adequately.

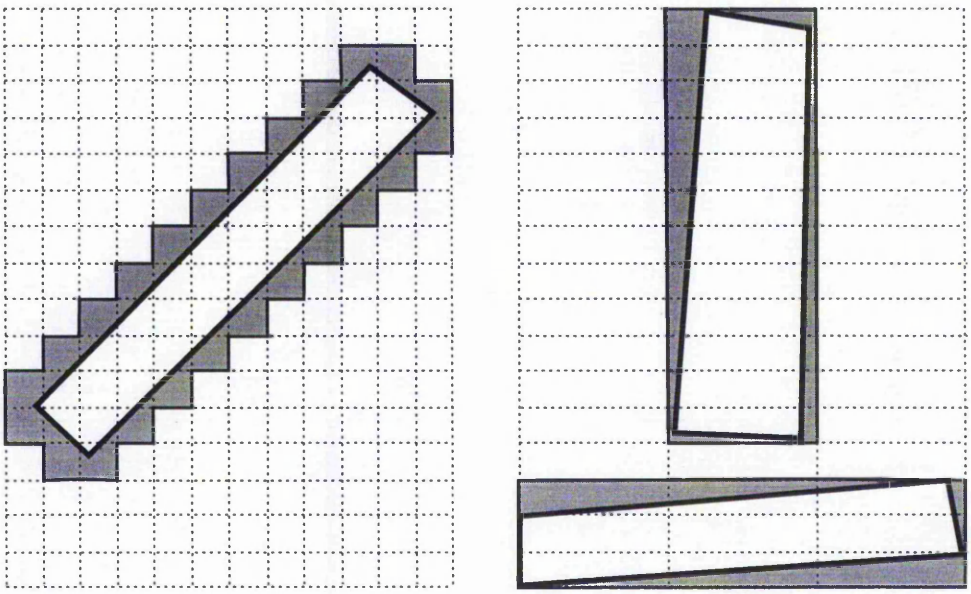


Figure 3.18 **The effect of pixel angle on choice of MIP map.**

3.6.4 Choosing the Optimal Number of Samples

From the above we see that the system would be most efficient if we could control the number of samples as a function of the size, shape and angle of the pixel footprint, rather than just keep it constant using the width. To do this we need a simple shape dependent parameter that can be easily calculated and which reflects the way in which the number of samples needed changes with all of these factors.

The ratio of area of ideal pixel to that of actual sampled pixel would provide a fair measure but would require far too much calculation to be usable in the present situation. It is possible, however, to find a simple parameter that behaves in an equivalent way. To make this calculation, we approximate the warped pixel to be a thin parallelogram and ignore what happens at the ends. This assumption is certainly accurate in those cases where a large number of samples need to be taken. It follows that we will never misclassify such a case. If we make a mistake which causes *too many* samples to be taken then the consequence is only a slight decrease in efficiency, which is outweighed by the time saved in making the test itself.

If we consider an individual column in the pixel footprint in Figure 3.17, then the area of the ideal slice is lh and that of the sample actually taken is $l(h+d)$. For the pixel as a whole these values become Lh and $L(h+d)$ respectively. The ratio of the areas is thus:

$$R = \frac{(h+d)}{h} = 1 + \frac{d}{h} \qquad \textbf{Equation 1}$$

Now if there are n columns in the pixel it follows that $d=(H-h)/n$ and so we have:

$$R = 1 + \frac{H}{nh} - \frac{1}{n} \qquad \textbf{Equation 2}$$

so we can choose an acceptable value for R and then derive n by the equation:

$$n = \frac{1}{(R-1)}\left(\frac{H}{h} - 1\right) \qquad \textbf{Equation 3}$$

this is reasonably tractable since it only requires the total height of the pixel and a sample of its "thickness" in the middle. If the equation is used in this form then there is a problem where H≈h (i.e. roughly square footprints) since a value of $n=0$ can result (or even a negative value of n, given the possibility of rounding errors). This is unacceptable since the minimum number of columns required is 1.

To prevent this from happening with a margin of safety, the equation is thus modified to read:

$$n = \frac{1}{(R-1)}\left(\frac{H}{h} + 1\right) \qquad \textbf{Equation 4}$$

This modification avoids negative or zero values for n and only becomes an influencing factor for roughly square footprints since for non-square footprints, the ratio of H / h is significantly greater than 1 and hence becomes the more dominant part of the equation. Equation **4** has the advantage that it allows the minimum value of n to be controlled by R and it introduces no extra computation compared to the original form (

Equation 3). To save computational time this quantity could be estimated on a per polygon basis. However, it is most effective to calculate it for each pixel as we have done in our experimental programs. We have found that it gives very good discrimination when used in this way. Quite large values of R can be chosen to give qualitatively excellent results without compromising speed. The improvement obtained by calculating n on a per-pixel basis compared with simply choosing a value for the polygon is shown in Figure 3.19. To illustrate this effect clearly it is necessary to use a rather lower number of samples than normal. The left hand side of the image uses a fixed n value of 2 whereas the right hand side has been created using a similar average number of samples but with n being

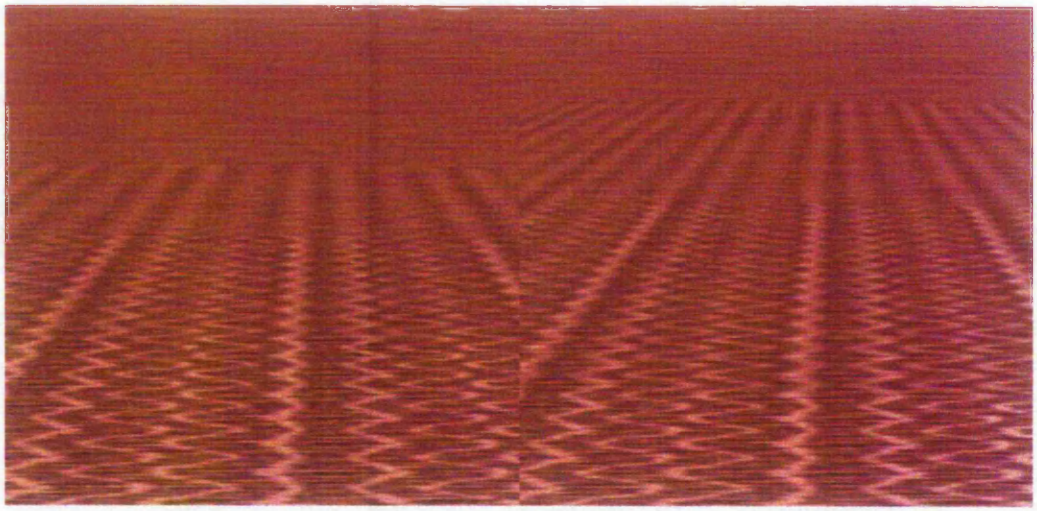


Figure 3.19 **Comparison between the effect of a fixed value for n (left hand image) and individual calculation for each pixel (right hand image). The average number of samples per pixel is 12 in both images.**

3.6.5 Performance Measures and Comparison with Other Methods

Section 3.4 provided a short taxonomy of published algorithms for anti-aliasing texture. Two of these algorithms will be used as a basis for comparison, namely, Adaptive Precision and Footprint Assembly.

In estimating the performance likely to be achieved by any of these methods, a number of different cost factors need to be considered. The true performance in a given situation based on user requirements can then be estimated by taking appropriate notice of each of them as the situation demands.

We can identify the following different performance criteria for texture antialiasing systems.

i. Computational cost.
ii. Number of samples needed per pixel
iii. Bandwidth to texture memory
iv. Table size

A comparison based on computational cost at the level of individual arithmetic operations would require an optimal implementation of each algorithm. Since all the algorithms in question are quite complicated, this would be difficult to do with any certainty. The result would also be machine dependent and, given the rate of technological change, time dependent. In contrast, the slowest moving performance factor in recent years has been off chip communication. For example, PC main memory bus speed has improved by only a factor of 3 between 1989 and 1999, while processor speeds have gone up by a factor of 12. This makes the number of texture samples required to generate a pixel a significant performance factor since it determines the number of independent memory accesses that will be required. It will also give an estimate of computational cost, since all the algorithms have a processing requirement that is proportional to the number of samples.

Of course the *number* of accesses is not the only factor to be considered. The *size* of each data item is also important because this will determine the total *bandwidth* that will be needed for this traffic. Using bandwidth as a measure of performance can be deceptive however, since different pixel packing and unpacking strategies will result in differing levels of performance. The best strategy therefore is to keep both measures separate until the hardware details are known. Consequently, although TPMM results have been presented, bandwidth information is also included for completeness. Note also that the effect of both of these numbers will, in turn be modified by any caching of samples which is taking place, but this effect is very sensitive to the cache size relative to the size of the texture map.

3.6.5.1 Comparison with Glassner's Algorithm

Glassner's paper [GLAS86] explores a number of different techniques for refining Crow's summed area table algorithm. Many of these proposals are not particularly practical since they require multiples of the full summed area tables to be stored. Glassner's most practical method is similar to TPMM in its sampling patterns for long thin pixels (although the number and size of samples differs). It is therefore possible to directly compare the two at a theoretical level, without the need to compare results visually.

Glassner's method improves on the rectangular approximation of the pixel footprint used in Crow's technique by subtracting further rectangular pieces from it. Because each piece is a rectangle, its contribution can also be evaluated using the same summed area table.

Both TPMM and Glassner's methods require a rather larger version of the texture map than standard MIP mapping or basic (non-anti-aliased) texture mapping, although the summed maps do not need to be quite as large as some have suggested [SCHILL96]. For a 256x256x8 bits per colour texture pattern a texture potential map could theoretically require 16 bits per colour and a summed area table as many as 24. However, this worst case is rarely met with practical textures and provided the colour values are calculated relative to an average value (which can be calculated separately for each column in TPM with little extra cost) 13 bits will almost always suffice. Indeed, if a small amount of compression of dynamic range is accepted this can be reduced to 12. The required table sizes for each of the techniques are summarized in a table at the end of this section.

When comparing sample count and bandwidth, the range of possible pixel footprint configurations makes a direct discussion of the general case quite difficult. Because of this, the simple cases when small values of n suffice will be examined first and then with the limit of large n, in which simplifying approximations may be made. The behaviour between these two points can then be inferred. In the simplest case Glassner's method reverts to the basic summed area technique and simply requires four samples per pixel. TPMM also requires a minimum of four samples in general since each pixel will span two columns. However, the summed area table samples require 16-24 bits per colour each whilst TPMM needs only 12-16 bits per colour. In addition the TPMM approach is, at this

stage, already able to follow a diagonally oriented pixel to some extent since the two pairs of samples can be vertically displaced relative to one another. In the horizontal direction, the sampling methods are rather different and hence a direct comparison is difficult. Superficially it might seem that the summed area method has an advantage in that its sampling window can be positioned at the resolution of the original texture map whereas TPMM is limited by the fixed sampling points of the current MIP map. However only helps when the texture pattern contains a frequency component with a particular phase relationship to the display pixel grid. This frequency component will then exhibit aliaisng if the image moves.

For large numbers of samples the same simplifying assumptions can be made about the pixel ends as were used in the discussion of the "R" parameter above. For TPMM the number of samples required for a given "R" factor is double the value of "n" as given earlier by Equation 3. For Glassner's method, consider a section from the middle of a diagonal pixel as shown in Figure 3.20.

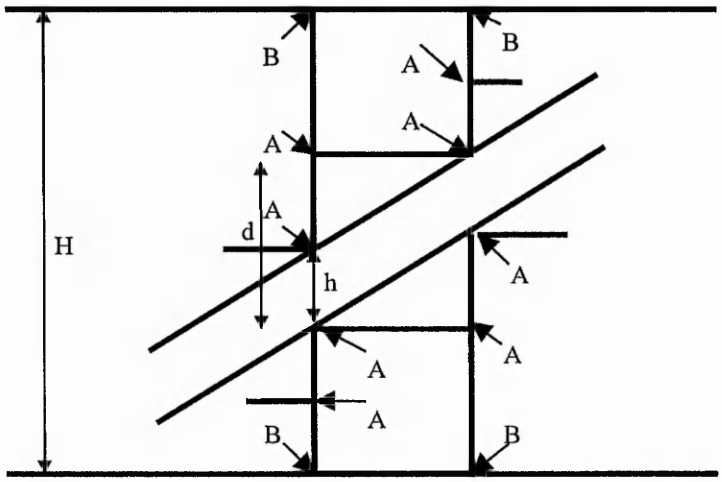


Figure 3.20 The sampling of a pixel using Glassner's adaptive precision technique.

The variables H,h and d have the same meanings as with the TPMM case. Glassner's method requires samples to be taken at the points marked A and, naively, two at each point marked B. However, these multiple samplings will cancel, leaving a total of 8 samples. These samples are of course shared between two columns, so the final result is 4 samples per column. This means that for a given "R" the number of samples will be four times the value of "*n*" given by Figure 3.20. Each of these samples will require 54 bits to be retrieved

for a 24-bit colour texture. TPM needs 39 bits per sample on the same assumptions. TPMM thus has an asymptotic advantage of a factor 2 for number of samples and better than 2.5 for bandwidth when compared with Glassner's method for similar image quality. For the less critical pixels, this will fall away progressively to equality for sample count and an advantage of 18:13 for bandwidth.

The above analysis was based on the situation for individual pixels. However, the existence of a discrete set of MIP maps in TPMM means that a demanded value of R and n will not normally be achieved exactly. Consequently, if a given *maximum* value for R is requested then on average the value of n actually used will be 50% larger than that given by the equation. This will result in a 50% reduction in the advantage of TPMM over the adaptive precision method if a "worst case" performance criterion is used.

3.6.5.2 Comparison with Footprint Assembly Algorithm

Another method that has been proposed to solve the shallow angle problem is the Footprint Assembly technique. As discussed in section 3.4, Footprint Assembly uses a standard MIP map table, but instead of allowing the scale to be determined by the largest dimension of the projected pixel the smallest dimension is used and multiple samples are taken to avoid aliasing. This technique can be used with single sample, bilinear or trilinear MIP mapping. These techniques are compared with TPMM in terms of image quality, number of samples taken and memory bandwidth.

Experiments have been performed using texture potential MIP mapping and Footprint Assembly Mapping. In each case the image created was of a complete "ground plane" of texture. The view of this ground plane was varied in two ways. Firstly, the angle of the plane itself relative to the line of sight was varied between 0 (looking straight down at the surface) and 1.5 radians. Since the angle of view chosen was 0.14 radians, this corresponds to the angle at which the point at infinity becomes just visible at the top of the screen. Any further rotation would simply add extra empty space at the top of the screen. This angle will henceforth be referred to as the α angle since it corresponds to a rotation about the x-axis in most conventional viewing coordinate systems. The second angle is that of the texture pattern itself, rotating in its own plane. This is referred to as the β angle.

The performance of the methods needs to be assessed in terms of image quality in those domains where significant differences can be detected and in terms of speed as an average over a representative range of different possible configurations. The qualitative differences under the same conditions can be observed in Figure 3.21 as distinct visual boundaries changing from good to blurred and from blurred to non-visible.

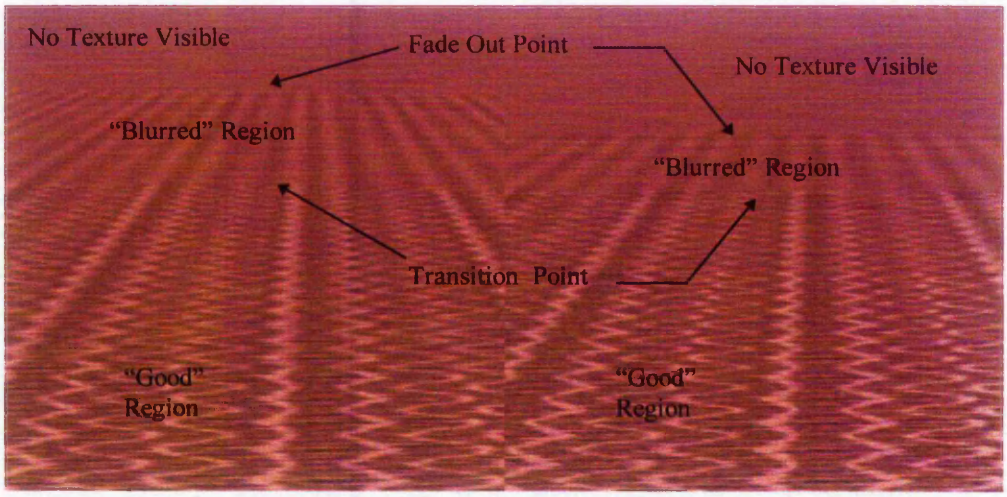


Figure 3.21 **Comparison of Texture Potential MIP Mapping (left) with R=5.76 and Footprint Assembly (right) with Nmax=4.**

The critical set of angles described previously creates long thin projected pixels that lie diagonally across the texture map. To make visual comparisons between the two methods unambiguous, implementations without interpolation between the different MIP maps were used for both algorithms. This creates clear discontinuities in the image in both techniques. Comparisons can be made between the methods by matching the positions of these discontinuities. The left hand side of Figure 3.21 shows the results of TPMM using an R-value of 5.76 at the angle specified above. For comparison the right hand side shows the results of Footprint Assembly mapping using bilinear interpolation and a maximum MIP map sample count of 4 at the same set of angles. In the foreground of each half of the image, there is a region where the texture is clear and well defined. At some point in the distance, this gives way to a more blurred effect and finally the texture fades out altogether. These features are annotated in Figure 3.21 and are present in all the images created. Two sets of comparisons were done. One set is based on the matching between the two algorithms of the point of transition to the initial blur. The other is the same except that it uses the final fade out point.

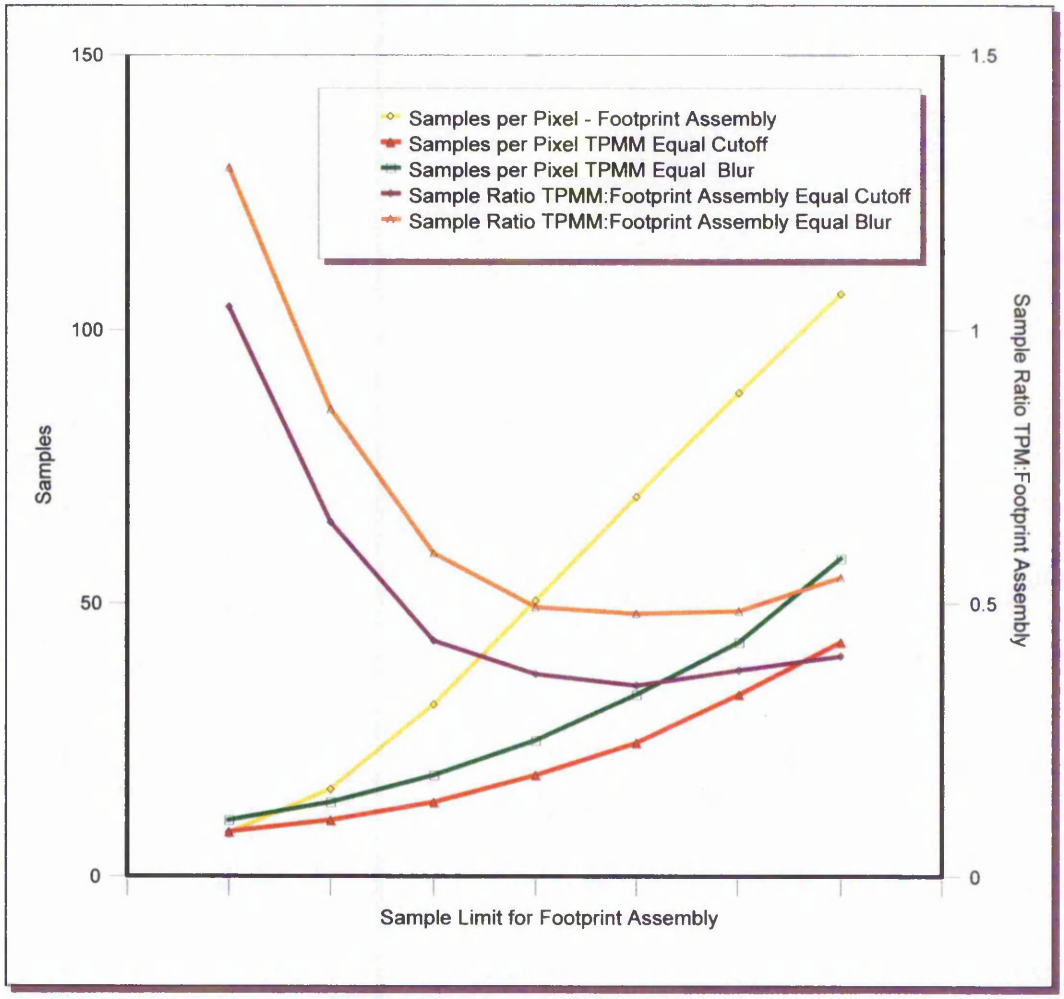


Figure 3.22 **Comparison of the number of samples required for Texture Potential MIP mapping and Footprint Assembly at varying levels of image quality.**

Figure 3.22 shows the relative number of samples required for each of the methods over a range of different levels of image quality but with the quality matched in each case between the two methods as described above and illustrated in 3.21. Two sets of curves are shown, reflecting the two comparisons made.

Overall, the results indicate that TPMM represents a better compromise between image quality and sample number/bandwidth for this kind of extreme situation. This advantage is most pronounced where footprint assembly has a sample limit in the range 8 to 32, corresponding to an average sample count in the range 10-100. This is also the most likely region in which either technique might be used. It is also important to consider the performance over the range of possible angles. The results in Figure 3.23 and Figure 3.24 use the "quality point" represented by N=16, R=3.38. Figure 3.23 shows the number of

samples required over a range of β values but at the critical α value. Footprint Assembly requires 50 samples throughout this range. The number of samples required by Footprint Assembly mapping is constant here because it is determined only by the projected pixel's shape. Conversely, that required by TPMM is reduced away from the critical β value since here the relationship between the projected pixel and the texel raster is taken into account. This can only increase the advantage of TPMM over Footprint Assembly. Note that this curve is not quite symmetrical about π/4 since the pure TPM method used for narrow pixels is slightly more efficient than the MIP mapping used for wide ones. In addition, Figure 3.24 shows the variation in sample requirement against α whilst holding β fixed at its critical value. Here the bandwidth required by Footprint Assembly mapping also reduces away from the critical region - but less steeply than for TPMM.

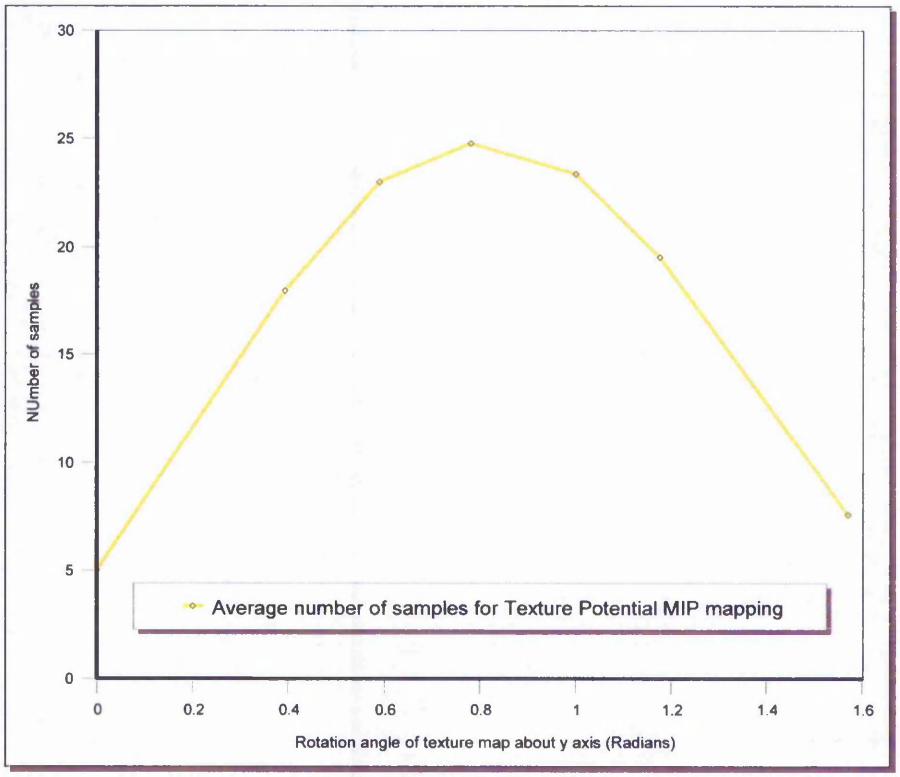


Figure 3.23 **Number of samples required by Texture Potential MIP mapping versus angle of rotation in the texture plane (β) R=3.38.**

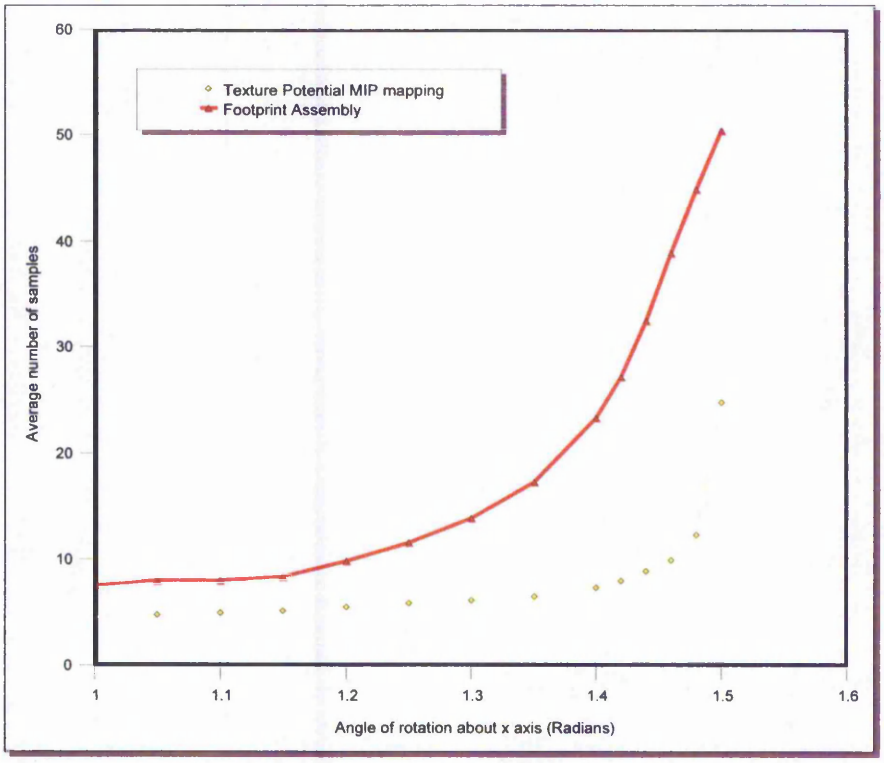


Figure 3.24 **Number of samples required by Texture Potential MIP mapping (R=3.38) and Footprint Assembly (N=16) versus angle of rotation of the texture plane α.**

These results clearly establish the greater efficiency of TPMM compared to Footprint Assembly with bilinear interpolation for static images.

The process of matching the transition points blurring and cut-off between the outputs of the two algorithms is of course a subjective one. In order to minimise the level of subjectivity within this process, several graphics experts provided independent assessments of where transition points occurred, from which average matches were made and sample counts deduced. The alternative to this process of course is to keep the level of performance fixed between the two algorithms evaluated the differences in quality observed. Figure 3.25 provides three image pairs showing TPMM results on the left and Footprint Assembly on the right using the number of samples. To obtain the outputs of Figure 3.25a, a total of 8 samples per pixel are required. Figure 3.25b requires 14samples and Figure 3.25c requires 30 samples per pixel.

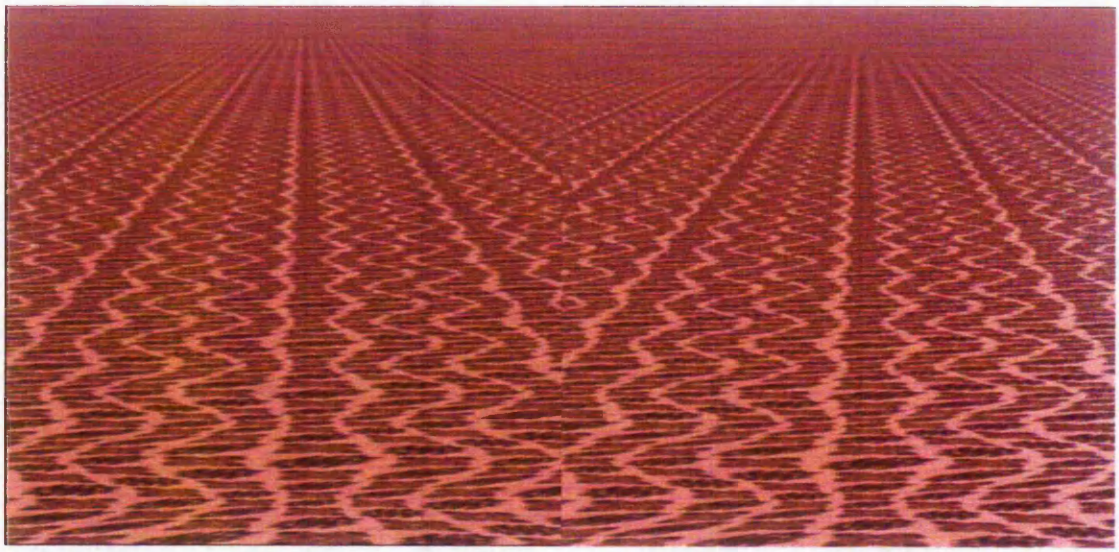


Figure 3.25b **The results of TPMM (left) and Footprint Assembly (right) using 8 samples.**

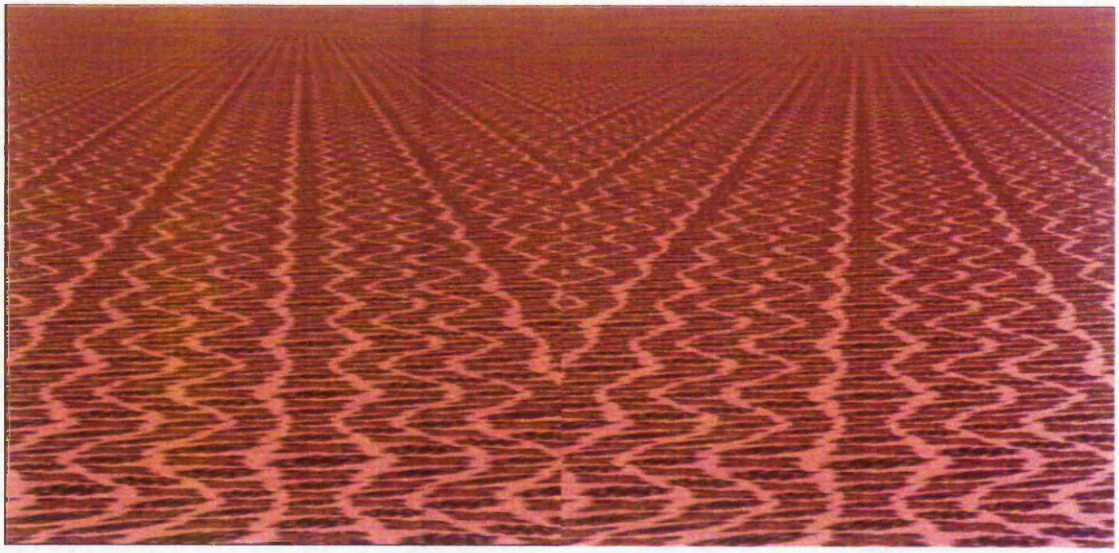


Figure 3.25b **The results of TPMM (left) and Footprint Assembly (right) using 14 samples.**

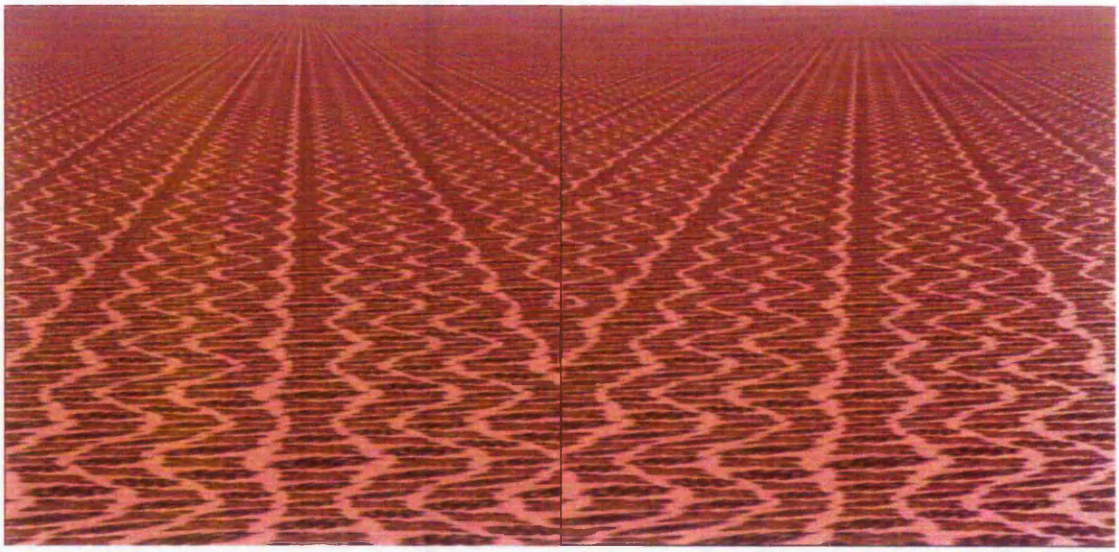


Figure 3.25c **The results of TPMM (left) and Footprint Assembly (right) using 30 samples.**

These results clearly indicate that the transition points discussed occur noticeably later in TPMM than in Footprint Assembly when the same performance levels are used. We can comfortably infer then that TPMM provides better quality output when compared with Footprint Assembly.

The bilinear interpolation within a layer in conventional (and Footprint Assembly) MIP mapping is a measure that prevents aliasing phenomena that would otherwise arise from the sharp transition between one texel and the next. If the surface is very close this aliasing would manifest itself as a patchwork of squares which, on close inspection, have jagged edges. There is room for difference of opinion as to how this problem *should* be resolved since clearly there is a lack of information in the texture map as to what happens at scales finer than one texel. The naive interpretation of constant colour over each texel produces the result as described above but without the jagged edges. The linear interpolation approach normally used in MIP mapping gives rise to a texture function that is continuous (C^0) but has discontinuous derivatives (not C^1). It is possible to demand a greater degree of continuity - but the interpolation required is rarely implemented at present since it is computationally expensive. In conventional MIP mapping this aliasing problem can appear at all scales and is particularly prominent when texture patterns are used which contain sharp edges (high frequencies). One situation in which this problem shows up is when such a texture is presented at a shallow angle. In this situation bilinear interpolation is

a texture is presented at a shallow angle. In this situation bilinear interpolation is essential with conventional MIP mapping if aliasing is to be avoided. Footprint Assembly MIP mapping allows the possibility that the extra samples used for bilinear interpolation could be used to increase the value of N, instead with improved results for static images. Unfortunately, the problem can also appear in the form of a scintillation effect with moving images and this happens irrespective of the angle of the textured plane. It seems therefore that the extra burden of interpolation cannot be avoided with Footprint Assembly if good quality results are desired.

These same problems also need to be addressed with TPMM. In this case, because of the asymmetry of the algorithm, the two directions need different treatment. In the MIP map direction this can be done by scaling the contributions of the end columns of the pixel according to their width. This approach has no effect on number of samples or bandwidth required. A completely general treatment requires the summed direction to have the interpolation applied also. In the summed direction, the problem only occurs with nearby surfaces because the texture map is always used at maximum resolution in this direction and so when the projected pixel is large it is finely sampled. If this effect is considered a problem, it can be dealt with by sampling the two neighbouring entries at each end of each column and interpolating between them. This will double the number of samples and bandwidth required but need only be done for certain nearby surfaces - which are otherwise undemanding in terms of required sample count. The impact on both average and worst case performance for TPMM will thus not be very great.

If we consider bandwidth rather than sample count then the results are slightly less unfavourable to Footprint Assembly since it deals with eight bit samples rather than the twelve bits (or slightly more, depending on the texture map size) required by TPMM. The sample count advantage of TPMM is about a factor of two at the critical set of angles and is larger at many other orientations so the situation should remain favourable when the bandwidth measure is used instead.

There remains the question of texture map size and here it has to be admitted that TPMM does require a larger amount of memory than either Glassner's method or Footprint Assembly. However, it does not require an excessive amount as do, for example, the brute

force approach of separate MIP maps or summed area tables for each possible angle of the texture pattern.

3.6.5.3 Memory Requirements

Typically, each MIP map layer in TPMM requires 1.5 times the memory of the original texture pattern. The multiple layers require a further factor of two, resulting in a factor of about 3 overall. It may be possible to use a factor 4 scaling between tables at some cost in sample count. In this case, the overall factor would be about 2. This compares with a factor of 1.33 for conventional or Footprint Assembly MIP Mapping and a factor 2 to 3 for Crow's or Glassner's method. Table 3.2 summarises the necessary sizes for the various methods described in the text. All the figures are based on a 256×256 texture map. The range of values given in the first row reflect the differing requirements that can occur with any of the summed area type methods depending on the content of the texture map. In the case of TPMM, there is a further variation that comes from the possibility of utilising MIP maps with a scaling factor of 4. Glassner's multiple table method allows a very wide range of possibilities depending on the required quality. The figures given are for the most basic version with just two tables.

Texture mapping method	Relative table size (minimum)	Relative table size (typical)	Relative table size (maximum)
Basic and Brute force integration	1	1	1
MIP mapping and Footprint assembly	1.333	1.333	1.333
Potential map	1.5	1.625	2
Summed area and Glassner's adaptive method	2	2.25	3
Potential MIP map	2	3.25	4
Glassner's multiple table method	4 (2 tables)	4.5 (2 tables)	6 (2 tables)

Table 3.2 Table of texture map sizes required by the different methods.

3.7 Conclusion

It has been shown that Texture Potential Mapping provides high quality anti-aliased texturing without demanding significant computation. It also has been shown how the Texture Potential Mapping Algorithm can be modified to keep the required sample count within reasonable limits by using the principles of MIP mapping in the non-summed direction. The resulting algorithm has a favourable combination of sample count, table size and quality compared with competing algorithms such as Footprint Assembly [SCHIL96] or the Adaptive Precision method introduced by Glassner [GLASS86]. The results of the algorithm are intermediate in quality between current real time hardware systems and what can be generated offline. As such, it is a candidate for implementation in future real time hardware.

The remainder of this thesis focuses on high performance hardware solutions for real-time computer graphics that embody the new techniques presented thus far into a flexible and scalable multiprocessing environment.

CHAPTER 4
MULTIPROCESSOR GRAPHICS ARCHITECTURES

4.1 The Need for Multiprocessing

Achieving a high degree of realism in 3D applications requires a significant amount of processing power. If we wish to enable realistic visual effects *without* visual artefacts, we need to employ the techniques described in the previous chapters of this thesis. A single processor performing this role for large datasets at high resolutions has an upper limit on what achievable if an acceptable frame-rate is to be sustained. The remaining chapters of this thesis explores ways in which graphics processing can be distributed amongst multiple processors and later presents a scheme of autonomous cellular graphics processing.

Modern graphics hardware has to cope with a throughput of tens of millions of texture-mapped polygons per second. Most hardware solutions on a PC offer this on a single dedicated chip containing up to 70 million transistors. Their speciality tends to lie in the acceleration of the rendering phase of the graphics pipeline only, using optimised scanline based techniques whilst leaving geometry-processing phase to the host processor. Hardware vendors are beginning to question this approach as the demand for better realism grows. Nvidia's Geforce chipset such as the Geforce2 ultra [NVID2000], is a good example of this change in which transform and lighting responsibilities are freed up from the host processor providing polygon rates of up to 30 million triangles per second. A further step is to parallelise 3D graphics processing by distributing data and/or tasks across multiple processors. An example of this is offered by the WildCat range of PC accelerator cards from 3Dlabs [3DLA2000]. These cards allow for multiple graphics pipelines to work in parallel. In the higher end graphics market, systems such as the Onyx 3000 series from SGI [SGI2000] offers very good scalable 3D graphics performance but with a significantly lower performance to cost ratio when compared with the previously cited examples.

The trend in graphics hardware design illustrate that future graphics solutions must offer longer-term architectural solutions using multi-processing in order to keep up with the pace of change with respect to graphics requirements.

There have been several different architectures that have been proposed and implemented over the years that take advantage of parallelism in both the geometry and rendering domain and these will initially be discussed along their respective advantages and disadvantages before the new scheme developed as part of the research is introduced.

The 3D rendering tasks can be split up into 3 main component tasks, namely database traversal, geometry processing and rasterization. On a simple system without sophisticated effects, the geometry processing phase is the most computationally intensive of these and requires floating point operations in order to perform geometric transformations on primitives in object space. The rasterization stage, which primarily consists of integer-based operations on primitives in screen-space, is less intensive. Furthermore, sophisticated effects such as high quality texturing will also be required and will incur significant speed costs on a single processor. Therefore, in order to achieve high performance with high quality in real-time, we must distribute the compute intensive tasks and/or their assigned objects over several processors.

A multiprocessing graphics architecture can be described as a cluster of geometry processors and a cluster of rasterization processors that are connected via an appropriate set of communication paths. The communication paths convey primitives, which are initially specified in object co-ordinates, from one set of processors to another, redistributing the primitives (or portions of primitives) where appropriate. The central problem with all multiprocessor architectures lies in how to allocate primitives to processors and redistribute them onto the screen during the rasterization stage.

The two basic forms of multiprocessing consist of *pipelining* and *parallelism*. A computation may be *pipelined* by partitioning it into stages that can be executed sequentially in separate processing elements (pipelining should not be confused with the standard graphics pipeline, which does not require multiple processors). A computation may be *parallelised* by partitioning the data into portions that may be processed independently by different processing elements [FOLE90]. Figure 4.1 illustrates the distinction between the pipeline and parallel multiprocessor schemes.

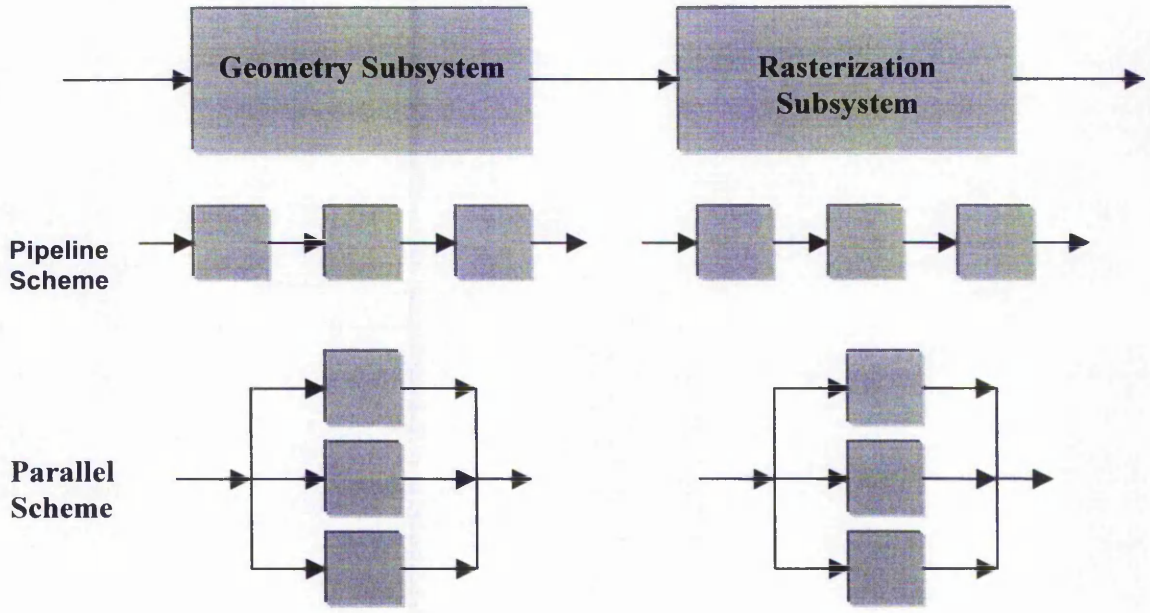


Figure 4.1 **Pipelined or parallel architectures can be employed.**

The ideal multiprocessing graphics architecture would be able to cope with a wide range of application domains, such as:

- Architectural walkthroughs
- Flight simulation
- Medical imaging
- Molecular graphics
- Computer-aided design
- Entertainment

Each of which require a particular suite of graphical effects, scene complexity and resolution in order to allow for any degree of realism. Therefore, in order to meet the needs of current and future 3D graphics requirements we must ensure that parallel graphics architectures are:

- **Real-time capable.**
- **Flexible.** *The system should be algorithm neutral.*
- **Scalable.** *Extra processors will proportionately extra performance.*

For a true multi-processor graphics architecture, scalability is a necessary condition, since any system that allows algorithm neutrality will ultimately suffer in terms of performance compared with its specialised counterparts.

To the author's knowledge, the architectures that have been implemented so far do not truly embody both flexibility and scalability combined. In order to determine a multi-processing architecture that meets all three of the above goals, we must ensure that the type of architecture chosen does not constrain the range of algorithms that may be used and furthermore, the choice of algorithm should not affect the rest of the design. For example, the choice of shading algorithm should be independent of the hidden surface algorithm used. Distributing intensive algorithms by task amongst several processors leads to a pipelining scheme that will accelerate graphics but will ultimately intertwine the processes involved. This therefore implies that any use of algorithm-based parallelism should not be considered. Thus, if we eliminate algorithm-based parallelism as an option, we are left to choose from some form of object-based parallelism or output-data based parallelism. A variety of flavours of object and screen based parallel schemes have been implemented over the years with varying degrees of success. These schemes will now be reviewed.

4.2 Screen Subdivision Methods

Screen space subdivision methods consist of parallelising the rasterization stage of the graphics pipeline by assigning a fraction of the pixels of the screen to each processor. This approach leads directly to the problem of how the screen should be partitioned. This is an important factor since if processors adopt a bulk-partitioning scheme by assigning contiguous blocks of the frame-buffer, there is a distinct possibility that load imbalances will occur when most of the graphical scene is transformed to only a portion of the screen. Thus the number of primitives in the busiest region would determine the frame rate of the system and a small region size per processor would have to be allocated in order to minimise this problem.

An alternative approach is to interleave the partitioning of the frame buffer so that an improved balance of workload is achieved. Figure 4.2 illustrates this approach and shows a more even balance across the screen. The example shows one possible configuration, in which interleaved memory locations are indexed via a lookup table using a key based on the 'real' pixel locations (in this case 'real' adjacent locations are separated by 2 pixels). This scheme is more efficient than the contiguous region approach in that system performance depends more on the total number of primitives.

A	B	C	A	B	C	A
D	E	F	D	E	F	D
G	H	I	G	H	I	G
A	B	C	A	B	C	A
D	E	F	D	E	F	D
G	H	I	G	H	I	G
A	B	C	A	B	C	A
D	E	F	D	E	F	D
G	H	I	G	H	I	G

Figure 4.2 **Diagram showing the memory arrangement for Interleaving**

The method of distributing work based on areas of the screen gives rise to an architecture

in which the sorting of primitives takes place *in-between* geometry processing and rasterization. Thus a sorting network is needed to connect the parallel geometry processors and parallel rasterization processors, since the primitives on any geometry processor may fall anywhere on the screen and may well be too large to be solely allocated to an individual processor. This network must take transformed primitives in screen space and determine which region (or regions) they affect, and convey them to the appropriate rasterizer. Figure 4.3 shows a schematic of the inter-processor communication.

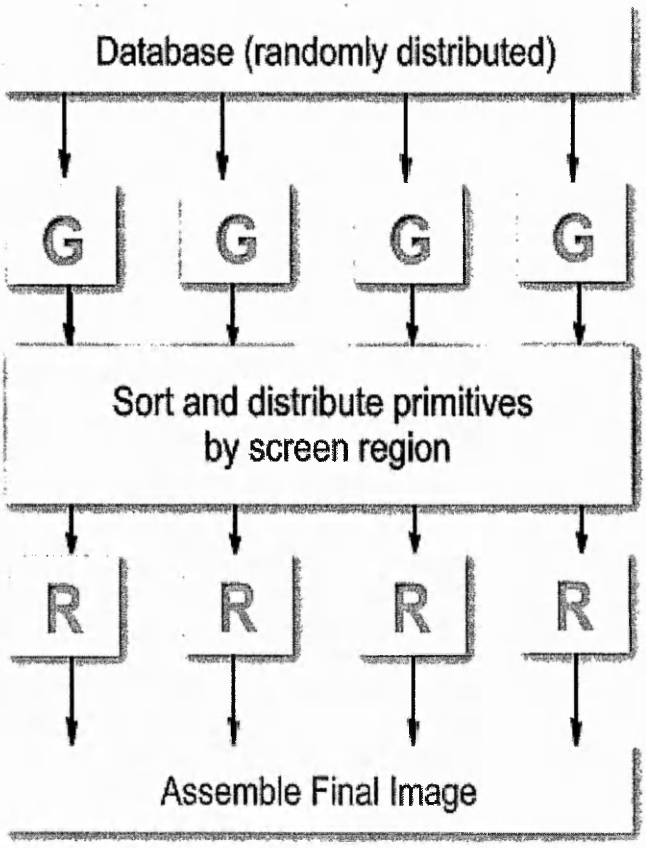


Figure 4.3 **Schematic of Screen-Space Subdivision Architecture where G =Geometry process, R =Render process.**

The network must communicate on a global level, since it receives its primitives from all of the geometry processors and must send them to all of the rasterizers. It also follows that a high bandwidth is required, as a description of the entire dataset must be transferred between the geometry processors and rasterizers between each frame.

The stages of rendering in parallelised image space architecture, using a contiguous

memory arrangement, generally proceed for each frame as follows:

1. **Transform primitives.**
2. **Sort image space primitives by screen region.**
3. **Transfer primitives to rasterizers using a high performance, high bandwidth network.**
4. **Process each screen region in parallel.**
5. **Collect and assemble image fragments in an image buffer for display.**

Such a scheme is not inflexible in terms of the order in which primitives are sorted and transferred. We could for instance, allow the processors to perform the sorting operations themselves and apply a search-on-arrival scheme. Furthermore, if an interleaved system of screen partitioning is adopted, we could send sub-objects, such as scan-lines instead of the object as a whole. It should be noted that when an interleaved memory arrangement is employed, the later stages of the above procedure differ slightly in that each rasterizer must handle all of the primitives, rather than just a subset that is relevant to an assigned region of the screen.

Screen space subdivision methods have several attractive qualities such as:

- Object primitives distribution methods are independent of how rasterizers are assigned to screen regions.
- Each rasterizer handles all of the primitives in a screen region, so the rasterization method is unconstrained.

However, this approach also leads to the following disadvantages:

- It requires a global communication network, which must transfer all primitives between geometry processors and rasterizers between frames.
- Bandwidth requirements for the network are high and scale linearly with performance.
- Complex software is required to implement sorting mechanisms
- Load imbalances can occur between rasterizers in a bulk partitioning system when primitives are unevenly distributed over the screen.

- Its latency is high in that all primitives must be sorted before rasterization can finish.
- Interleaving makes it very difficult to incorporate custom rendering effects and overlays. Software based interleaving provides some degree flexibility in this case but will significantly reduce performance even for simple bit block transfers. Caching algorithms and multiple back buffers are also difficult to implement.

Screen subdivision architectures, although intrinsically flawed, have been put into practice with respectable performance output. Example architectures are Pixel Planes 5 [FUCH89] and the Silicon Graphics Power Iris 4D/240GTX [AKEL89].

4.3 Pixel Processing Systems

One of the earliest implemented Parallel architectures is the pixel processing system developed by the University of North Carolina called Pixel Planes [FUCH81]. This involved many processors (SIMD – Single Instruction, Multiple Data) that were assigned to individual screen pixels and were essentially a form of logic enhanced memory. Thus, Pixel Planes made use of frame buffer subdivision in order to accelerate graphics processing.

A prototype system was developed in 1986 [EYLE88] and used a design similar to that of the VRAM chip except that the one-bit ALUs and associated circuitry replace the role of the video shifter.

The performance of the pixel planes system did not solely rely on parallelism however since each pixel processor would have to perform all the operations required for scan conversion independently, leading to inefficiencies from redundancy. The scheme devised to overcome this problem makes use of a linear expression tree in which linear expressions are evaluated in parallel. Thus for every pixel (x,y), floating point coefficients A,B and C are input to evaluate expressions of the form $F(x,y) = Ax + By + C$. Since many rasterization calculations make use of linear forms, each pixel receives its own value of F in its own local memory. Figure 4.4 shows the schematic for the Pixel Planes 4, logic enhanced memory chip.

One of the main advantages for this system is its simplicity since it does not need to be concerned with ordering or distributing tasks for each processor and hence there is no pre or post processing required. However, pixel-processing systems suffer from a severe lack of algorithmic flexibility and are not cost effective for appreciably high-resolution graphics requirements.

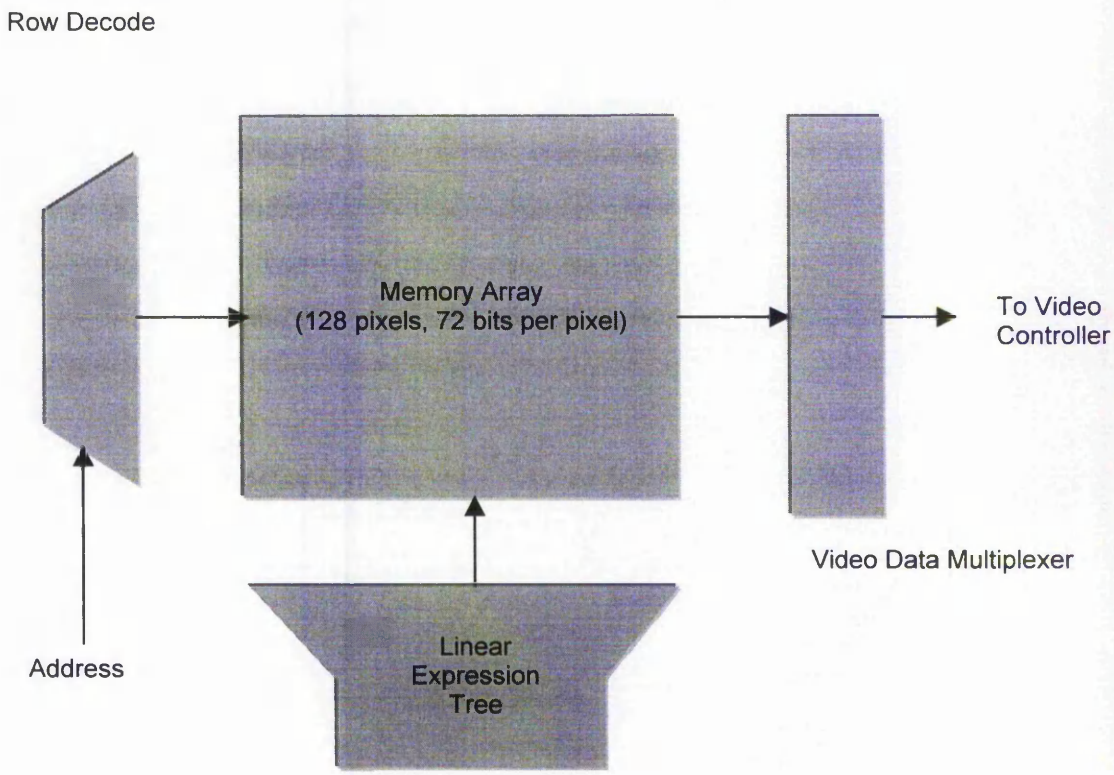


Figure 4.4 **Schematic of the Pixel-Planes 4 logic-enhanced-memory chip**

4.4 Object Based Parallelism and Image Composition

The object-parallel family of parallel architectures parallelises the inner loop of image-order algorithms (which tend to be scan-line). These architectures process multiple primitives in parallel, so that final pixels may be generated more rapidly. The usual approach is to assign primitives to a number of object processors that enumerate the screen pixels in scan-line order and store colour and depth values for their assigned primitives. The pixel streams from each of the object processors can then be combined to produce a single stream for the final image.

One of the first commercial implementations of a real-time, processor per primitive system was the General Electric's NASA II flight simulator [BUNK89] which proved to be a simple and appealing architecture with no defined limit on scalability. However, use of object parallelism alone tends to restrict the types of primitives that can be displayed and the types of shading algorithms that can be used. Furthermore, the sheer size of the rendering task in real-time graphics makes a purely input data based scheme difficult to construct unless the screen coverage of individual data items can be constrained in some way.

4.5 Image Composition Methods

The principle of image composition methods has been used in various forms for many years, particularly in the video industry, such as video overlay and chroma-key techniques. In this scheme, geometry processors and rasterizers are paired to form renderers, as in screen space subdivision methods. Each renderer is assigned a random portion of the primitive database and is made responsible for the entire screen, as opposed to only a portion of the screen. This means that each renderer computes a full screen image of its portion of the primitives. These images are then composited together in such a way that surfaces in one partial image hidden by those of another are eliminated. Figure 4.5 shows a schematic of the inter-processor communication and Figure 4.6 shows the distinction that is made between image composition methods and screen subdivision methods.

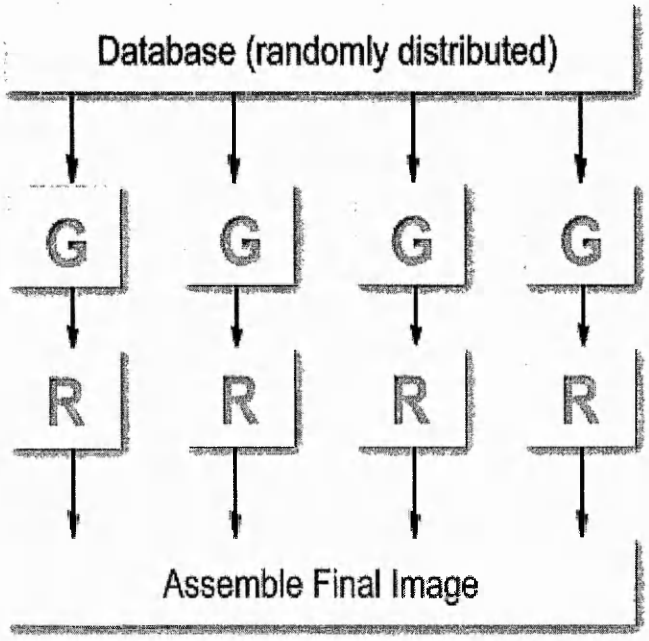


Figure 4.5 **Schematic of Image Composition Architecture**

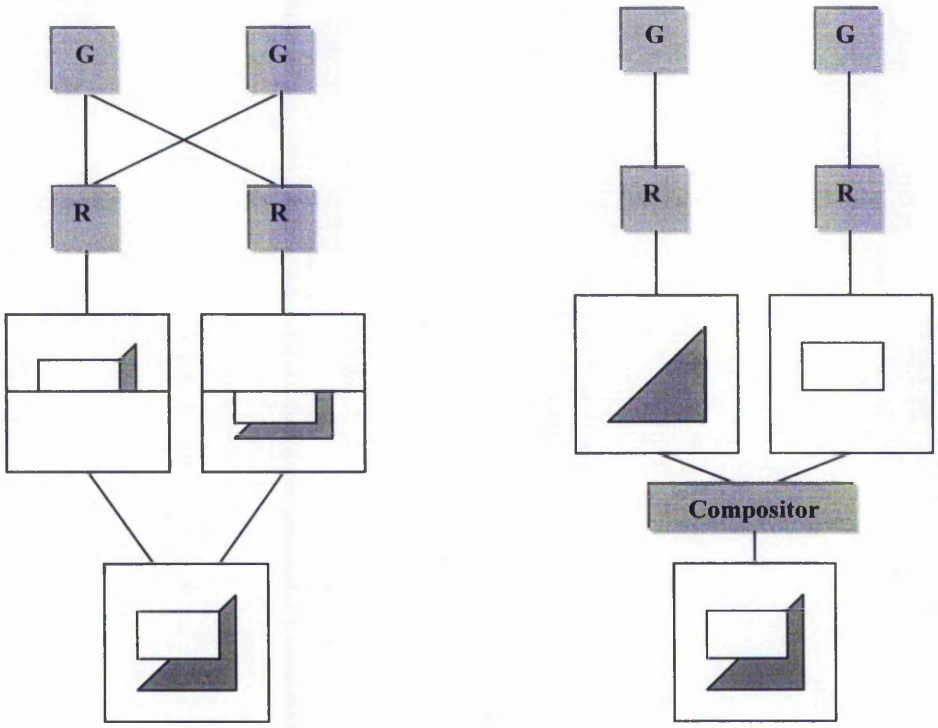


Figure 4.6 **Comparison of Screen Space Subdivision and Image Composition methods**

The main advantage of image composition techniques is that no sorting or redistribution of primitives is required in the renderers. Each renderer computes the image of its primitives as if it were the only one in the system. The sorting network in this architecture takes the form of a high-bandwidth network that composites images. The simplest way to do this is to composite images pair-wise. This results in either a binary tree or pipeline structure for the image composition network. Video scanout from each frame buffer occurs in the normal way, except that the contents of each z-buffer are scanned out as well. Scanout processes in each frame buffer are synchronised so that each frame buffer scans out the same pixel at the same time. Since the network must operate at high speeds to maintain real-time frame rates, special composition processors or compositors are required. However, the compositor implementation may restrict the kinds of rendering algorithms that the architecture can support.

An interesting property results from image composition architectures, since they can be scaled to an arbitrarily high performance by adding renderers and compositors. This is a result of employing a tree-based or pipelined composition network that can accommodate

an arbitrary number of nodes. Furthermore, renderers compute their sub-images independently.

The advantages of image composition architectures are summarised as followed:

- Renderers work within a simple context. They compute their partial images independently
- Load balance is automatic for geometry operations since each renderer can be given roughly the same amount of work.
- Communication is local and the bandwidth is constant. It is determined by the maximum rate in which images can be composited.
- The architecture is linearly scalable.

However, image composition has the following disadvantages:

- It requires a distributed display database.
- It imposes constraints on the rendering algorithms that may be used since the rendering method must produce pixels in a format suitable for compositing, and hence the flexibility of the system is reduced.
- It requires a very high bandwidth for the image composition network for communication between renderers.
- Up to a frame of pixel storage is required per renderer.

A good example of image composition is the PixelFlow system [MOLN92] which is linearly scalable and achieves up to millions of triangles per second. The PixelFlow system is composed of a series of renderers and shaders on a 256 wire back plane running at 132MHz. The renderers operate by sequentially processing 128x128 pixel regions of the screen. They scan out the region's rasterized pixels over the image composition network in synchrony with the other renderers. The Shaders load pixels from the image composition network; perform texture mapping and shading; blend subpixel samples for anti-aliasing and finally forward pixel values to the frame buffer. Figure 4.7 shows a block diagram of the prototype PixelFlow system.

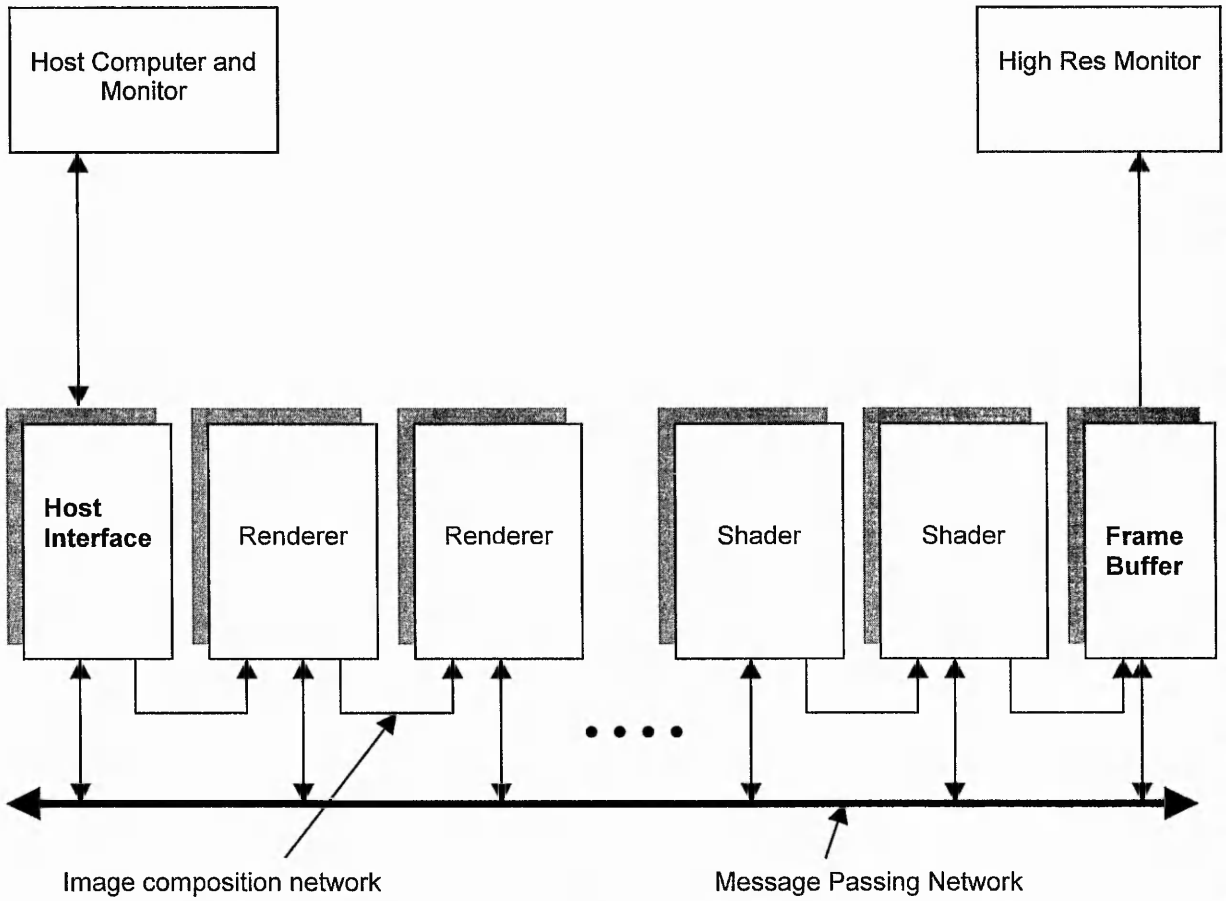


Figure 4.7 Block Diagram of the prototype PixelFlow System.

4.6 Summary

To summarise the strategies outlined in this chapter, we see that object parallelism alone is detrimental to our goal of flexibility since the screen coverage of transformed data objects needs to be constrained in order to get significant performance improvements. Furthermore, output based parallelism alone exhibits high bandwidth requirements and load imbalances can readily occur when a large proportion of the dataset transforms to a small fraction of the screen. The most promising semi-commercial solutions provided so far have been image composition based architectures but, as outlined previously, they do not provide a true flexible, scalable solution. It may be argued that no such solution can possibly exist: by increasing system performance, we must forfeit the generality of the problem and vice versa. The following chapter attempts to disqualify the previous statement by investigating an alternative hybrid architecture in which the author attempts to retain generality (i.e. flexibility) whilst achieving high performance.

CHAPTER 5
THE CELLULAR ARRAY ARCHITECTURE

5.1 Autonomous Processing / Localised Storage

Another approach to parallelising real-time graphics is to *combine* object-based and screen-based parallelism together. Such an architecture would speed up the early part of the graphics pipeline by using object based parallelism and then later recombine the final image from processors that are dedicated to working in different parts of the screen. Figure 5.1 illustrates the hybrid architecture, in which geometry processes are bound to rasterization processes.

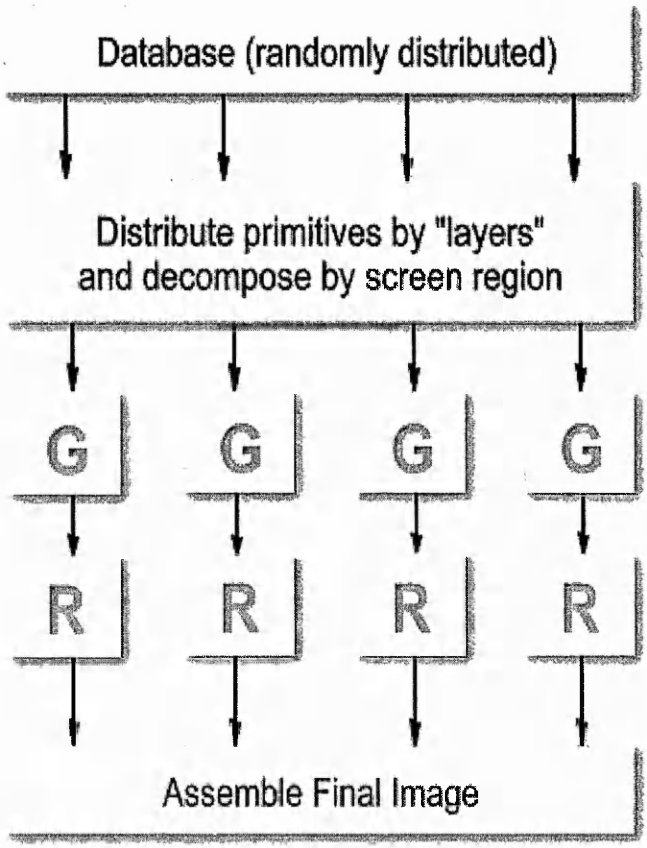


Figure 5.1 **Process schematic of hybrid graphics parallelism.**

To the author's knowledge, the benefits offered by this scheme have yet to be fully exploited. One of the main reasons for this is due to problems that arise in the first stage shown in figure 5.1. Here, some form of polygon distribution scheme is required that will dynamically balance out the processing load evenly throughout the processor network. The main goal of this chapter therefore, is to explore the potential benefits offered by this scheme, whilst offering new solutions that overcome the distribution problem.

Geometry processors and rasterizers are combined to form a rectangular array of autonomous processors so that minimum interaction is required between each processor. Figure 5.2 shows a schematic of the array, in which the horizontal direction refers to processors dedicated to different parts of the screen and the vertical direction refers to the primitives being processed. Here, the processor arrangement accommodates for three objects that can be processed in parallel. Each of these objects can be divided into four quadrants of the screen. The topology of such an arrangement has lead to the title of 'cellular array', which will be used to refer to the architecture throughout the remainder of the thesis.

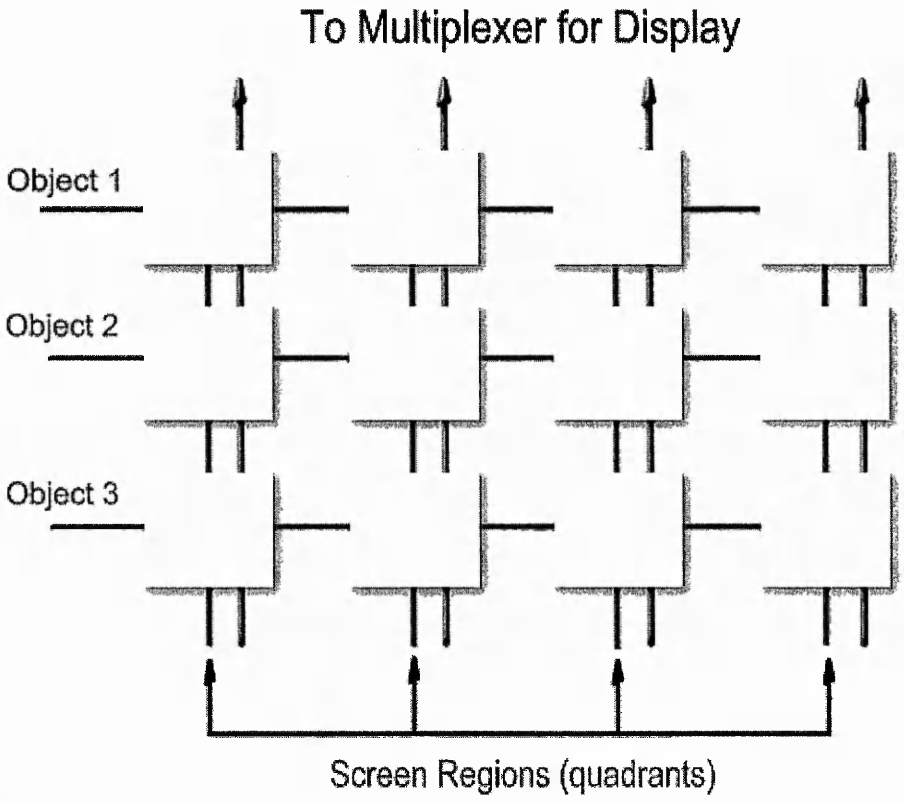


Figure 5.2 **Schematic of Cellular Array Architecture (objects are assigned to banks of general processors which are clipped according to screen region)**

The unconnected links at the top of the diagram go to the output hardware, which must contain some multiplexing capability in order to create a single image from the four separate parts. The links at the left-hand side are used to provide input to the processors from a host processor. The form of input data at its most basic level would consist of polygon mesh data that is distributed from the host processor to processor banks that subdivide screen space. The inter-processor links in the vertical direction need to allow RGB data to be created from their output at the frame rate of the system at least, and would therefore need a sufficient bandwidth to accommodate for this.

By combining both object and image space parallelism, we immediately simplify the form of intercommunications network required and decrease system bandwidth requirements significantly since each processor would only need to receive primitive vertex data as input and pixel data as output. Furthermore, object based constraints are lifted since the screen coverage of each object would now be reduced to manageable levels.

The potential advantages of cellular array processing are therefore threefold:

- It is flexible: Because each processor carries the *complete* set of graphics algorithms representing the graphics pipeline, any change to the system is simply a straightforward coding operation with no consequence to the hardware of the system.

- It is scalable: An increase in rendering performance is possible simply by increasing the number of processors in the horizontal direction when an increase in resolution, depth complexity or rendering sophistication is required. Conversely, a greater capability in terms of the number of objects rendered can be achieved by increasing the number of processors in the vertical direction (the number of banks).

- It is not driven by fleeting technology: The only major dedicated hardware required is the output multiplexer needed to reconstruct the final image. Such a device only requires upgrading with advances in output display technology and thus tends to

have a relatively long shelf life. Furthermore, the addition of processors to the array should only require code recompilation and the provision of an interface to the multiplexer.

- Bandwidth is reduced to a minimum: Since geometry processing and rendering are bound together for each processor, virtually all storage is localised so that each processor receives mesh data as input and provides pixel fragment data as output to the multiplexer.

However, there are two requirements that must first be met in order to ensure that implementations of the cellular array are both truly flexible and scalable.

- The cellular array network must be re-configurable. That is, it must be relatively simple to add new processors and processor banks to the network without requiring detailed knowledge of the system.

- The cellular array must have a balanced loading and be scalable. In order to ensure an even load balance in the array, the host processor would have to employ a rather complex distribution algorithm. This would result in an undesirable bottleneck on the system.

The following sections investigate how these requirements can be met.

5.2 The Recombination stage – FPGA Solution

Recombination logic needs to be able to adapt to different processor network configurations. If no such feature exists, then we would restrict the range of possible applications and would not meet our goal of flexibility and scalability. The cellular array therefore requires some form of re-configurable hardware to sort out the connection logic between processors. Field programmable Gate Arrays are a good candidate for this role since the combination logic itself is very easy to program and we have complete flexibility (even at run time) to reconfigure the cellular array. It is also possible to pre-program a variety of recombination algorithms onto the FPGA to perform the role of switch-able node-points between processors. Figure 5.3 illustrates the core component requirements for each cell.

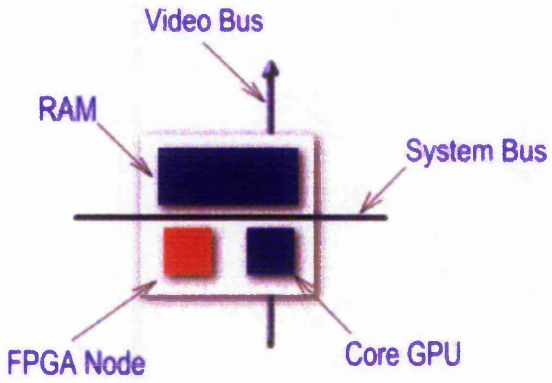


Figure 5.3 **Schematic of a cell indicating re-configurable networking nodes.**

Once a processor configuration is established, we must also consider the time at which recombination should occur. This is a critical factor since if it occurs too early, we must 'pass-on' data that may be required by other algorithms to finish the raster process. In this case, the constraints on algorithmic flexibility will dictate once more. Hence, the recombination should take place at the last minute so that data structures will merely be pixel related.

5.3 Polygon Caching – Reducing the Distribution Problem

The architecture, although attractive in many ways, does carry with it the legacy of load imbalances inherent in the distributed screen-based methods described in chapter four. One solution to this problem would be to pre-sort the database so that the distribution of objects is evenly spread with respect to screen region, combined with image based interleaved memory [Clark and Hannah 1980]. Such solutions however, will over-burden the host processor in the first case and restrict flexibility in the second. An alternative strategy is therefore required.

In the most extreme case, only a single processor within each bank of the cellular array would be active at any one time (i.e. all current polygons are clustered within a small region of the screen). A significant number of processor banks would therefore be needed in order for the system to maintain a satisfactory frame-rate (this case, although undesirable, is also probabilistically rare). Therefore, in order to keep every processor in each horizontal bank active at all times, each individual processor should be given object data on demand as opposed to waiting for the busiest processor in a bank to finish before allocation can occur again.

In order to meet this requirement, a per-processor polygon cache scheme is proposed, in which the host processor passes new object data whenever a processor in a given layer is idle, whilst simultaneously filling the caches of the other processors in that bank with the same object data for subsequent processing. Thus, each processor can reach its pertinent data as quickly as possible by clipping away irrelevant object space primitives (a relatively inexpensive geometric operation) without waiting for its neighbours to finish. The memory requirements for each cache elements consists of floating point vertex data, texture coordinates and material properties. This approximates to 64 bytes per entry for triangle mesh data. Figure 5.4 illustrates polygon caching using two cells. The left cell can process larger polygons (with respect to screen space) without holding up the right cell. Furthermore, the cache data can be used to retain estimates on load distributions between successive frames.

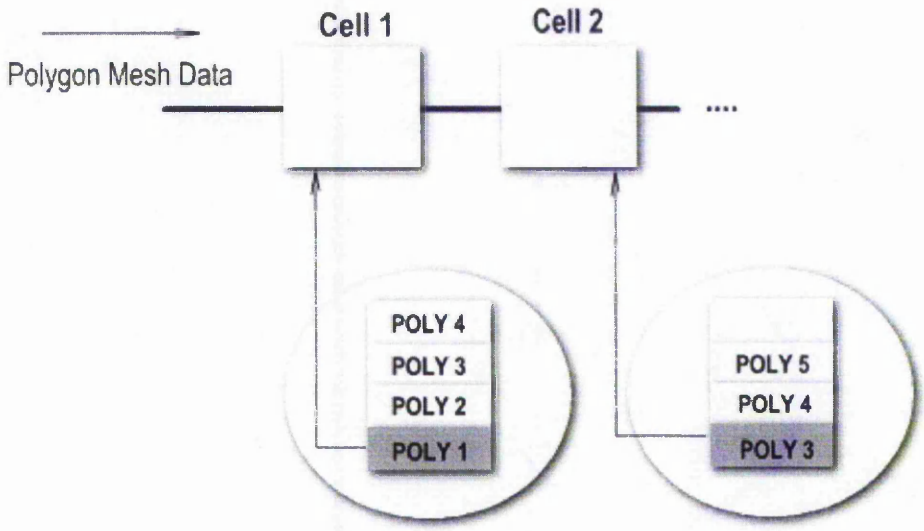


Figure 5.4 **Internal cellular polygon caches ensure that all cells are busy**

5.4 Hidden Surface Removal and Anti-aliasing

In order to support a wide range of applications (e.g. flight simulators, CAD, V.R. walkthroughs etc), the proposed architecture should be supplied with a basic set of algorithms in software that are neutral so that they not constrain the range of special effects that can be added in the future. This would allow the developer to build *onto* them, as opposed to re-writing entire applications from scratch. These algorithms should therefore be general in nature and should be used over a wide range of applications.

The most general of algorithms for rendering depend on the type of scheme and graphical effects used throughout the graphics pipeline (see chapter 1). In the case of the cellular array, we are considering a scheme that starts with database objects and results in screen pixel information. We must ensure then, that sufficient information about objects at the pixel level is maintained in order to facilitate the range of algorithms that deal with aliasing, shading, alpha blending, texture mapping etc. This is normally dictated by the form of hidden surface removal algorithm that is employed since information about surfaces needs to be maintained in order to compare them with new surfaces.

As mentioned in Chapter 2, the most sensible choice for high performance hidden surface removal is the Z-buffer since it makes no demands on the general structure of the other parts of the graphics pipeline. However, it does suffer from the fact that depths are only sampled at a single point and only a single value is kept from the contributions of surfaces to each pixel. Thus, more information about pixel contributions needs to be generated with minimal computational cost and retained in such a way that the overall system is not algorithmically constrained.

5.5 Anti-aliasing – Pixel Fragment Recomposition

The modified A Buffer described in chapter 2 is a good candidate for anti-aliasing since it retains the advantages of performance offered by the Z buffer and also deals with multiple pixel contributions to produce high quality results. Information about pixel contributions and depth extremities would be passed in the vertical direction of the array. However, since cellular processors are working with objects in local memory until multiplexing occurs, care must be taken regarding the combination of pixel fragments. We would therefore require some form of pixel assembler to combine multiple fragments of each pixel packet into a single RGB value for the pixel. This involves the composition of fragments in front-to-back order, taking into account how much of the pixel each new fragment covers. Figure 5.5 illustrates the logic required to make depth comparisons at the point of composition without anti-aliasing. By storing pixel based information locally to each processor, inter-processor communication is kept to a minimum and memory access conflicts are avoided.

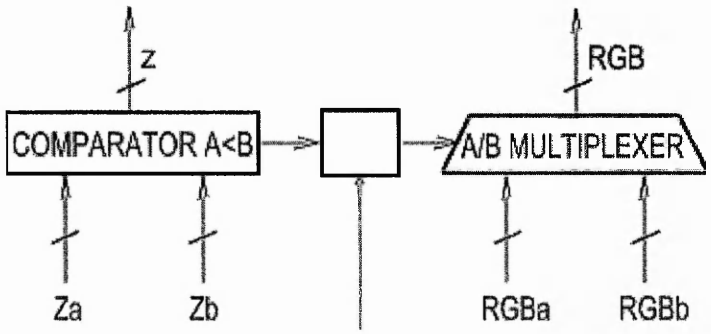


Figure 5.5 **Logic required for a simple z-buffer compositor.**

5.6 Cellular Array Experiments

5.6.1 Implementation Considerations

In order to ascertain how well the cellular array would perform using different configurations and datasets, simulation runs were performed on a single processor using the following assumptions as a basis for implementation:

1. The time taken to perform geometry calculations for each polygon with respect to screen region is approximately proportional to the single processor case. The basic geometry-processing phase of any graphics pipeline consists of camera/object transforms, lighting calculations, texture clamping and vertex/shade clipping to the viewing-frustrum. If we sub-divide individual polygons according to screen regions in parallel, then the difference in the time taken to perform this operation (compared to the non-subdivided case) will depend on the maximum number of new vertices created as a result of sub-frustrum clipping. For large datasets, the number of polygon intersections will be small compared to the total number of polygons. This approximation can therefore be used as a basis for knowing *when* processor banks should notify the host processor for more data in the simulation runs.

2. The optimal size of polygon caches can be approximated dynamically (this optimal size in this case defines how much of each cache the cells can use rather than physical size of the caches). This is a requirement since different scenes will require different limits on the cache size if optimal results are to be observed. However, there are several strategies to estimate this optimal size at run-time without burdening the host processor (these will be discussed later).

3. Memory storage for pixel, depth and texture data is localised to each processor. This is in fact a consequence of the architecture itself. Access conflicts are minimised and bandwidth requirements are not an issue until final depth buffer comparisons amongst each column of processors are made for display (see point 4).

4. The simulation results are independent of both the multiplexing technology used as well as the speed of the bus to the multiplexer. The reason for this assumption is to simply avoid obtaining results that depend on device dependent technology that will change over time.

5. Estimates as to *when* the host processor should allocate polygons to cell banks is achieved by cells calling back to the host on demand. If a given cell in a bank has finished rendering, it is either ready to retrieve another polygon from the cache (no call-back is necessary) or it is ready to obtain data directly from the host. In the latter situation, it is likely that the other cells in the given bank are still actively rendering. In this case, the host must also provide the same polygon data to the other cells in the bank by pushing the data into their respective caches. In a physical system, this would require a brief interruption in the execution of all processors in a bank in order to perform a memory access. These interrupts have been approximated to occur in a single cycle of the host.

5.6.2 Simulation Test Data

The characteristics of scene data must reflect not only the application environment, but also the constraints defining how users can interact with it. For example, ground based data in flight simulators will in general, bias polygon distributions to the lower half of object space with atmospheric data such as clouds being described and rendered using a different set of algorithms. Applications such as walkthroughs however, will consist of a combination of large planar surfaces encompassing complex hierarchical objects consisting of many polygons. Two distinct classes of datasets have used to represent these extremes. Figure 5.6 illustrates the two sample data sets used to simulate interior walkthroughs with the cellular array[1]. The bowling alley output represents a dataset that uses relatively large planar surfaces with some symmetry evident in the distribution. The Escher dataset however, represents a more extreme case with 'random' distributions of polygon densities throughout the scene.

[1] Line rendering is used here for clarity only – the actual output generated by the cellular array uses texturing and shading algorithms.

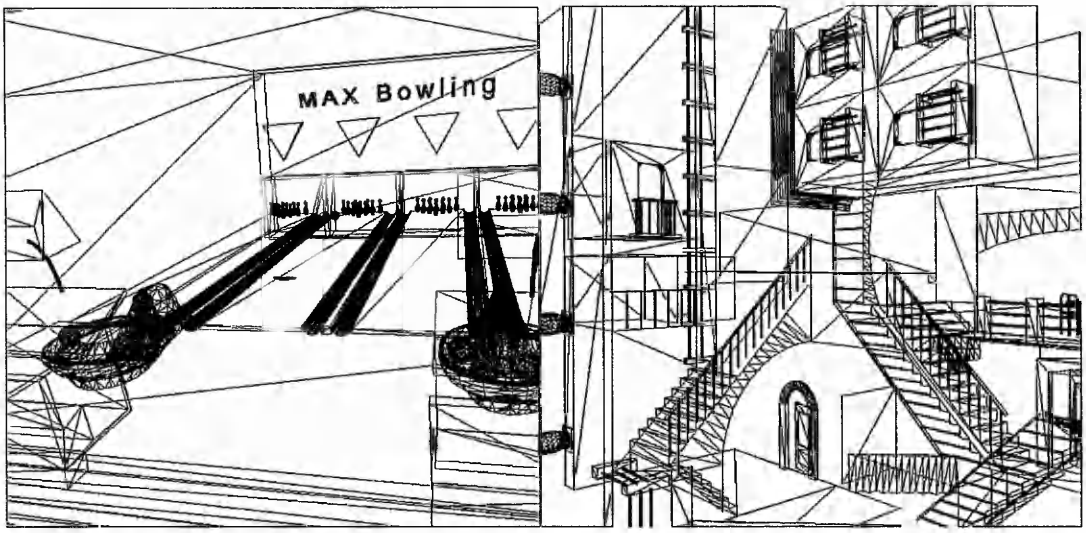


Figure 5.6 **Left: The bowling alley, consisting of 60777 polygons. Right: 'Escher's House', consisting of 10938 polygons.**

To simulate exterior scenes, an implementation of a fractal landscape generation algorithm was used. Fractal mountain scenes of increasing complexity were generated as test data to control both the complexity and distribution of objects within scenes. Figure 5.7 shows several such datasets taken from a total of seven that were used to generate results. The numbers of polygons generated for scenes in this case are 832, 3328 and 13312 polygons respectively.

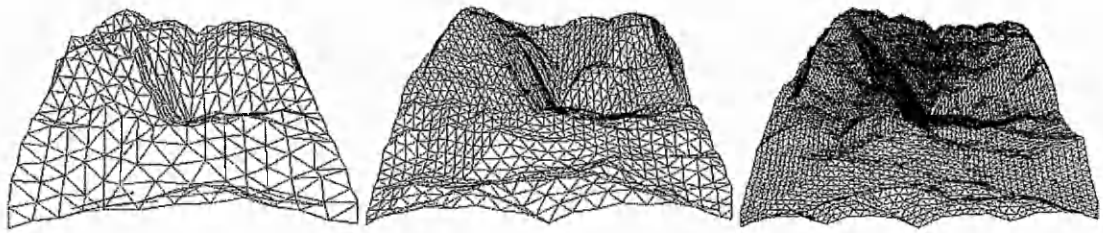


Figure 5.7 **Landscape data generated procedurally with increasing polygon count. Mountain 1: 832 polygons, Mountain 2: 3328 polygons, Mountain 3: 13312 polygons**

Figure 5.8 shows a typical test scene using shading and texturing. Several concentrations of polygons (the trees and the helicopter) are also included in order to generate additional loading bias in the data distribution.

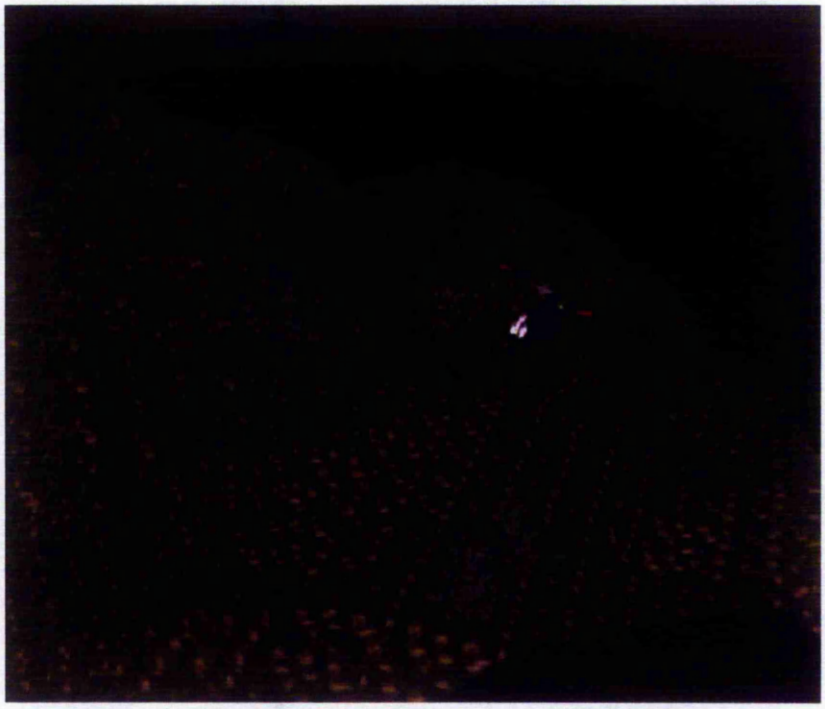


Figure 5.8 **One of the scenes used to generate flight simulator characteristics for cellular array simulation runs. This scene consists of 20 000 Gouraud shading, textured polygons.**

5.6.3 Measuring Cellular Array Performance

A typical performance measure for many hardware graphics systems is based on the number of Gouraud or Phong shaded triangles processed per second. Since such information would not be realistic without a hardware implementation of the cellular array, a *relative* performance measure is used against the results produced by a single processor performing the same task. This is based on the total number of clipping / render cycles taken to process the dataset. In the case of a single processor, this is approximately proportional to the total number of polygons and z-buffer accesses made (without sophisticated effects).

The following performance ratio gives us a relative measure of cellular array performance with respect to varying banks:

$N_{single} / (C \times N_{array})$

where N_{single} is the number cycles taken to generate a single frame on a single process and N_{array} is the number of cycles taken in an array using C processors per bank. The division by C of course will result in a performance ratio approximate to one in a worse case scenario using a single bank.

If we only wish to measure the relative performance of the effect of screen-based parallelism however (i.e. for only a single bank of cells), then the ratio of the number of the cycles taken for a single bank array to the number of cycles taken for a single process is used instead i.e.

$N_{single} / (B \times N_{array})$,

with B being the number of banks in the array.

Finally, to deduce the *overall* cellular array performance, the measure of

N_{single} / N_{array} is used.

5.7 Simulation Results

Figure 5.9 shows the performance results for varying numbers of processor banks using four cells per bank, over a single 20 000 polygon mountain dataset and with 3 different viewing orientations (as shown in figure 5.10). The distribution bias of polygons with respect to screen region shifts from the lower half to the upper half. Since depth complexity and resolution are fixed in this case, performance gains are most evident when additional banks are added, as opposed to increasing the number of cells per bank. The first performance measure given in the previous section is used to deduce bank dependency.

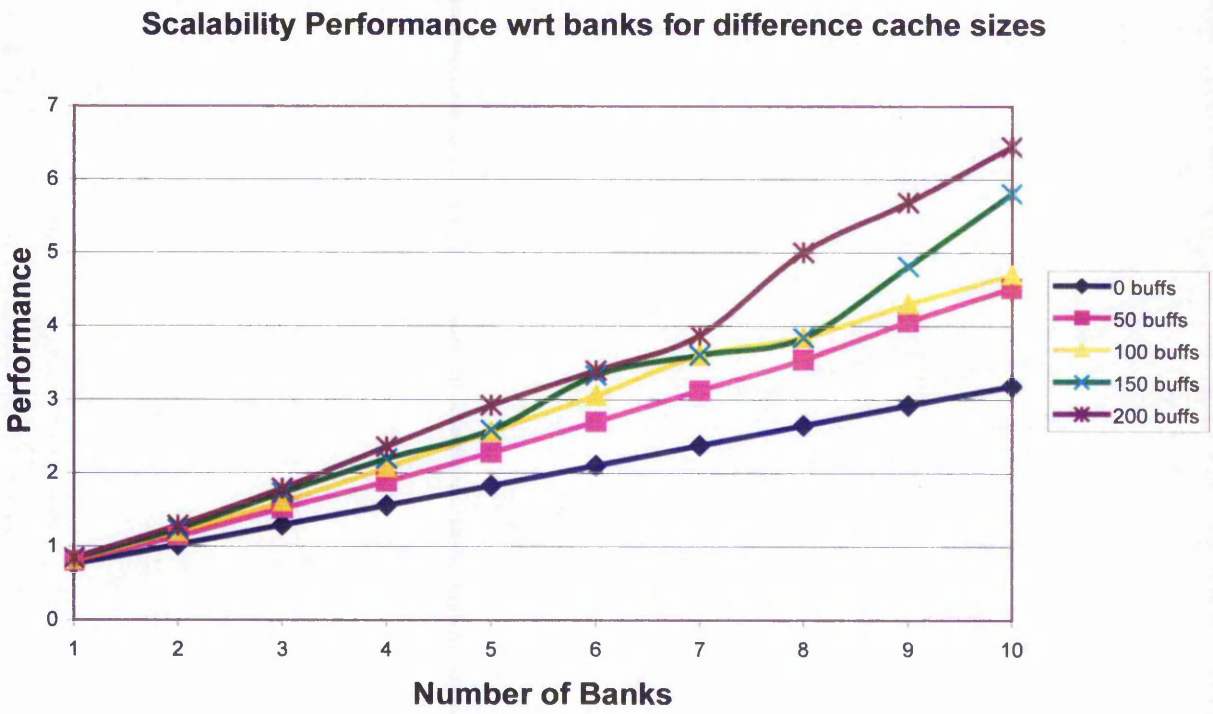


Figure 5.9 **Performance results of landscapes averaged over 3 different viewing orientations with respect to an increasing number of processor banks.**

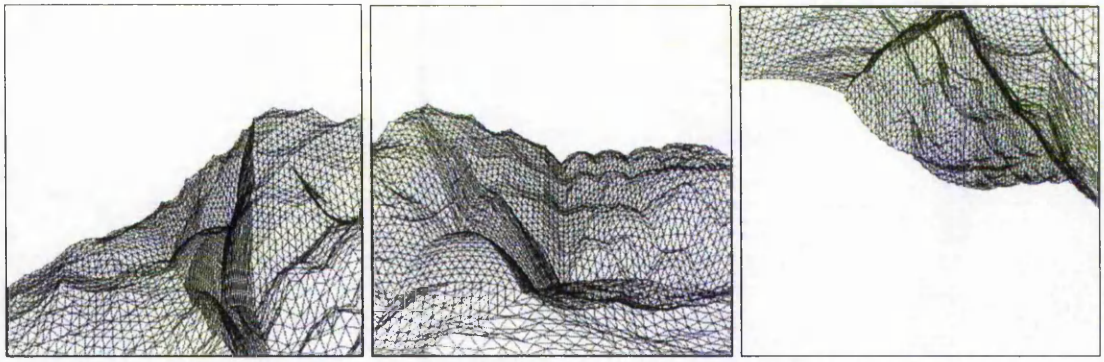


Figure 5.10 **A 20000 polygon dataset viewed from three different camera orientations.**

The scaling of performance with respect to polygon cache size displays approximate linearity [SHRU98], but becomes more unpredictable with increasing cache size. This is due to the distribution task of the host processor finishing too early, so that certain layers become 'bogged down' with the work of objects covering a large area of the screen (and thus filling up their caches too quickly) whilst other banks are dealing with smaller objects which take less time to process. In other words, the system has resorted to the original scene-dependence/loading problem highlighted in chapter 4. Therefore, care must be taken with respect to the actual size of the cache allocated to each processor if we wish to retain any predictability of the system.

From the performance data provided in figure 5.9, we see that a cache limit of 100-150 polygon entries would provide a good combination of linearity and high performance. The problem of unpredictable scalability can be therefore be reduced simply by controlling the global polygon cache size. This can be achieved by exploiting frame-to-frame coherence so that the appropriate global size of all polygon caches for consecutive frames can be estimated. This data can be derived from the results of the previous frame that was rendered. The cache size therefore, can be estimated by passing back the maximum number of polygons that are present in the caches of each cell after a frame is rendered. The host processor can therefore remotely call each bank with an average of these sizes before distribution occurs for the next frame (an inexpensive role to play).

Figure 5.11 shows how the gains in performance become saturated as we increase the buffer size. The same data and orientations were used as in figure 5.8. This effect depends more on the size of the dataset rather than its distribution. Hence, only an approximation of the upper limit of the cache sizes is needed for good results.

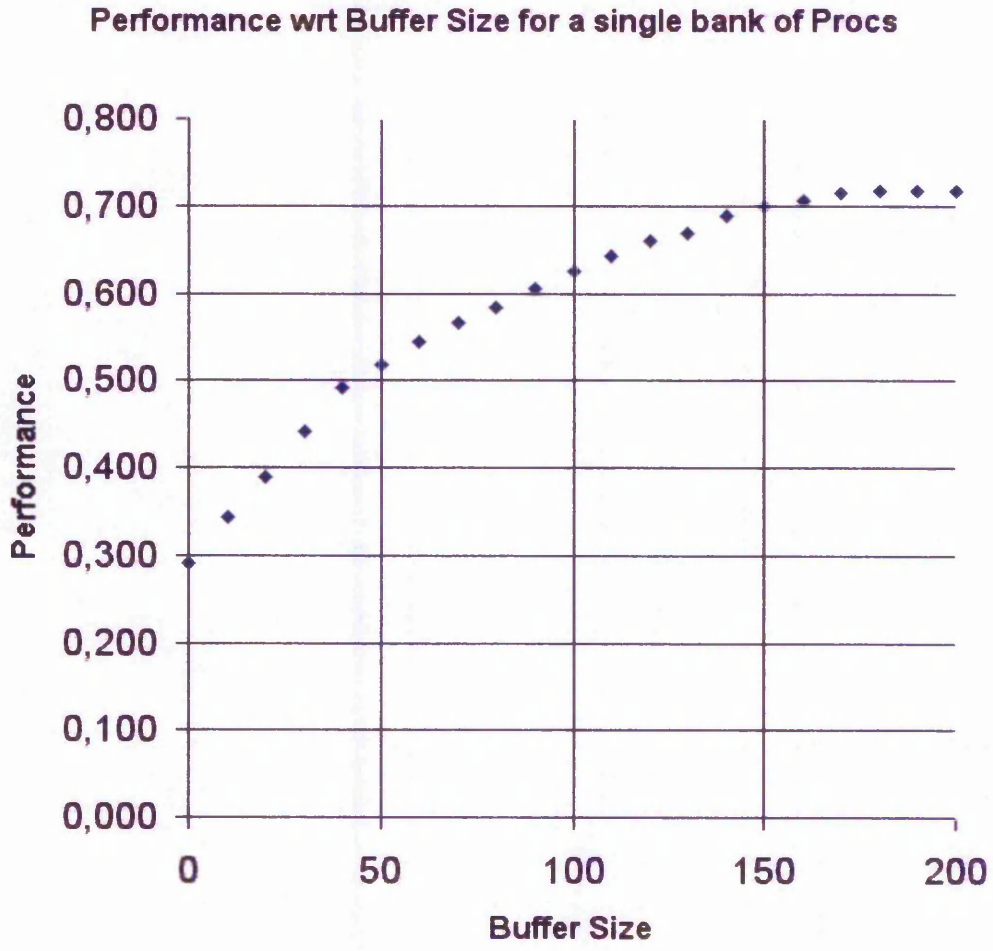


Figure 5.11 **Performance gains for different buffer-sizes with a fixed number of banks**

So far, only a limited range of test data has been used to illustrate the behaviour of the cellular array. The following tables provide an overview of how the performance varies for a wider range of dataset distributions when a good estimate on the optimal cache size is used from frame coherence. Table 5.1 again looks at the increase in performance with varying banks with an average cache size given based on 3 random viewing orientations of the datasets presented earlier. We see that performance boosts are not quite as significant for smaller datasets as they are for the larger ones. However, the bowling alley data still presents some minor problems for the array, particularly in the 4

bank case, where the increase in performance is not as great as expected compared with the 2 bank case.

Table 5.2 shows how the cellular array performs as the number of cells *within a single bank* is varied. The need for changing the number of cells per bank is more obvious than the previous case since it is less dependant on the particular application and more on the resolution of the final output display (in this case 768 x 768). Polygon caching is also used here to generate performance estimates based on the sizes estimated for table 5.1 (using the overall performance measure). The benefits of caching in this case are not as significant as in the multi-bank configuration since bank cells will take longer to bypass irrelevant polygons.

DATASET	Polygons	Average Cache size	1 Bank	2 Banks	4 Banks	8 Banks
Mountain1	832	100	2.60	5.08	5.24	8.36
Mountain2	3328	300	2.88	5.08	5.32	3.78
Mountain3	53248	600	3.56	6.96	13.36	24.8
Bowling Alley	60777	5000	3.92	7.32	7.6	14.12
Escher's House	10938	500	3.44	4.32	9.2	10.16

***Table 5.1* Performance results from 5 different datasets using an optimal cache size and varying number of banks with 4 cells per bank.**

DATASET	Polygons	2 Cells per Bank	4 Cells per Bank	16 Cells per Bank
Mountain1	832	1.83	2.90	2.96
Mountain2	3328	1.43	1.87	2.03
Mountain3	53248	1.27	1.66	3.14
Bowling Alley	60777	1.91	3.94	5.7
Escher's House	10938	1.79	3.46	3.95

***Table 5.2* Performance results from 5 different datasets using the optimal cache size and a varying number of bank cells for a single bank.**

5.8 Incorporating Parallel Potential Mapping

As discussed in chapter three, anti-aliasing schemes for texture mapping can easily 'bog-down' any graphics system when high quality, non-blurred results are required. As a result, a form of hardware accelerated texture mapping must be implemented in order to realistically meet these demands in real-time. This section provides a new mechanism for hardware accelerated TPM that can be integrated into the cellular array.

5.8.1 Hardware TPM

Texture potential mapping (in its most basic form) can be decomposed into four distinct stages:

(i) Transformation
(ii) Pixel Edge Set-up
(iii) Pixel Edge Tracing
(iv) Summation of Final Results

Implementing TPM and TPMM is a relatively simple task in software since the algorithms do not have to deal with 'special cases' that can easily overcomplicate the design. This therefore offers the opportunity to pipeline the latter half of these stages in to distinct hardware components as shown in figure 5.12. Pipelining these tasks provides only a partial solution though, since very large textures will still incur long processing times.

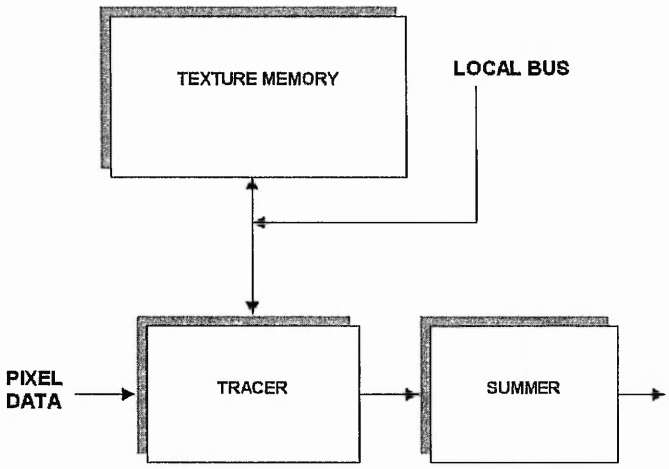


Figure 5.12 **Process schematic of TPM indicating the 'potential' for a viable hardware implementation.**

To fully accelerate TPM and TPMM, we must distribute multiple tracers and summers (i.e. texture processors) so that each is responsible for a particular region of the texture. This will ensure that render times will remain within tolerable limits, even when texture sizes become very large.

In order to devise an efficient scheme of TPM acceleration, we must ensure that both the waiting time for texture processors and communication bandwidths are kept to a minimum. To achieve this, texture processors should be left with as large a portion of the work as possible, even if this means that some duplication takes place. Since each processor is responsible for a fixed region of texture space, it follows that the region of screen space that they deal with will change over time as new frames are rendered. The easiest way to implement this is to pass an entire polygon worth of data to each texture processor and allow the processor itself to decide those parts for which it is responsible to process. However, texture processor responsibilities will grow out of proportion (almost to the same scale a scan converter), making at much more expensive to implement in hardware. A better solution is to let each texture processor make decisions based only on single scan-lines (as shown in figure 5.13). Thus, we must pass screen-based polygon spans to each processor along with the texture matrix.

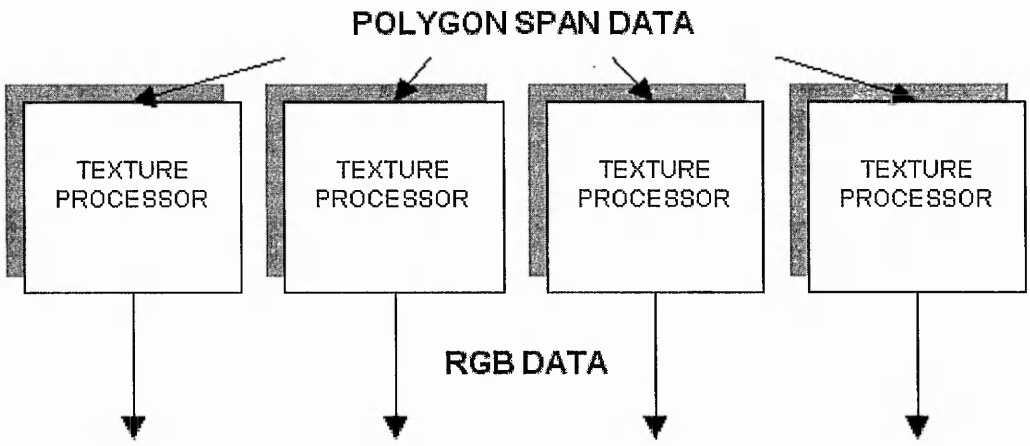


Figure 5.13 **One possible parallel texture configuration to distribute the texturing task (textures accessed on a different bus).**

A simulation of this scheme has been implemented in software to examine the performance gains possible. The following algorithm was used to generate simulation results.

```
For each polygon span
        Pass texture matrix and span extents to each texture processor
        For each texture processor
                Warp span extents to texture-space
                Clip according to assigned texture region
                For each warped pixel
                        Perform TPM for an RGB value
```

Since each processor will subdivide span data to regions that it is responsible for, the time taken to texture-map a given polygon span will depend entirely on the maximum length of the warped sub-spans in texture space (the value 'd' indicated in figure 5.14).

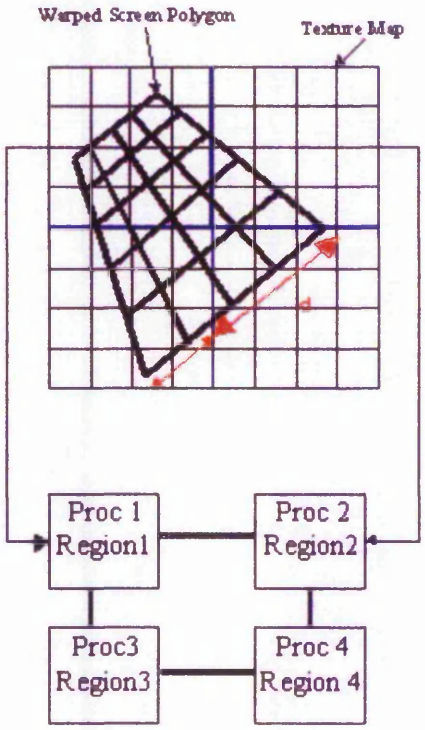


Figure 5.14 **Texture space parallelism splits up large textures and accelerates render times when polygon spans cross texture boundaries.**

Performance gains are most evident when large textures are used on large planar surfaces such as those used in flight simulators and interior walkthroughs. Benefits of this scheme are not so evident however when textures are 'wrapped' onto objects that consist of many micro-polygons. In this case, a single texture is clamped onto thousands of polygons resulting in ony a few texels per polygon. The case in which small textures are tessellated over large surfaces however, is easily dealt with by tessellating the texture into available texture memory before processing occurs. Figure 5.15a shows a mesh using shrink-wrap texturing of a 512x512 owl texture. Texturing with four texture processors accelerates the process by a factor of 1.5 when compared to a single processor. However, Figure 5.15b is textured on a per polygon basis using the same configuration with much a more acceptable performance gain of a factor of 2.3. The table below summarises the results of the average relative performance between the two texture schemes against a single texture processor.

TEXTURE WRAPPING SCHEME	Performance gain
Cylindrical Wrap	1.54
Planar clamping to polygons	2.35

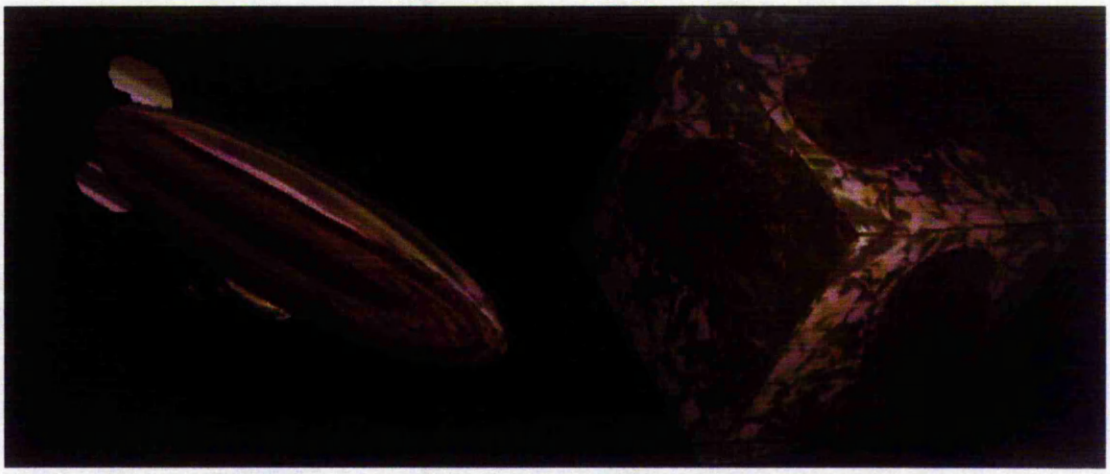


Figure 5.15 **The left image uses texture wrapping based on-object space geometry per object-mesh (the blimp) giving poor performance boosts when parallelised. The version on the right clamps textures over large polygonal surfaces in which texturing is significantly speeded up when parallelised.**

5.8.2 Hardware TPM and the Cellular Array

The problem now remains as to how the Parallel texture scheme should be linked in with the cellular array architecture [SHRU96]. There are two possible strategies that can be employed here, each of which has different implications regarding efficiency and complexity.

The first possible approach is to assign a texture processor to each individual cellular processor within each bank of the array to form a 'cubic' array topology (in concept only – in reality, they would be present on the boards of the cells themselves). This would ensure a simple communication network since texture processors can receive input direct from processor cells and output values along the same bus. However, the problem of redundancy may well occur when cells are busy in the middle of a process. The possibility of cells being idle (again causing redundancy) is minimal here, as empty cells will quickly fill up with small polygons. Figure 5.16 shows the arrangement of processors for this scheme.

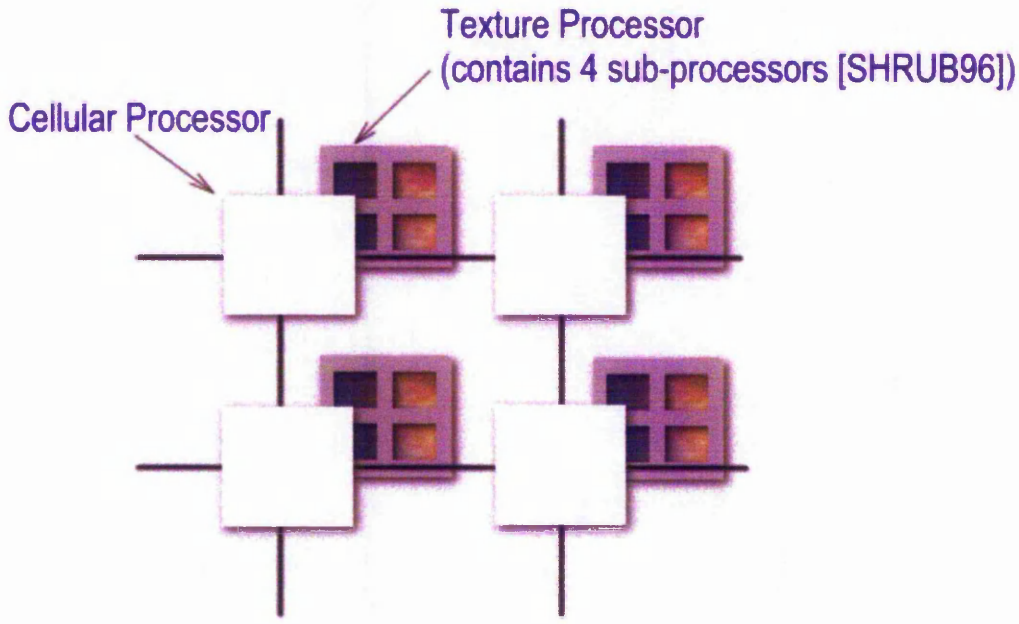


Figure 5.16 **Each cell in the array is linked to its own texture processor such that a 'cubic' cellular array is formed.**

Although this scheme has the advantage of simplicity, at must also be noted that memory requirements are significant. 32MB of texture memory per processor would provide a reasonable compromise between performance and cost, so that overloading the system/graphics bus at run-time is not a controlling factor on performance.

An alternative strategy is to adopt a processor-farming scheme in which a bank of texture processors is connected to all of cells in the array. In this scheme, a cell that requires texturing of its assigned object may access any of the idle texture processors in the bank. This ensures that texture processors work at peak efficiency, but there must be a sufficient availability of them in order to avoid un-textured objects "queuing up" for processing. Figure 5.17 shows how processors will communicate in such a scheme.

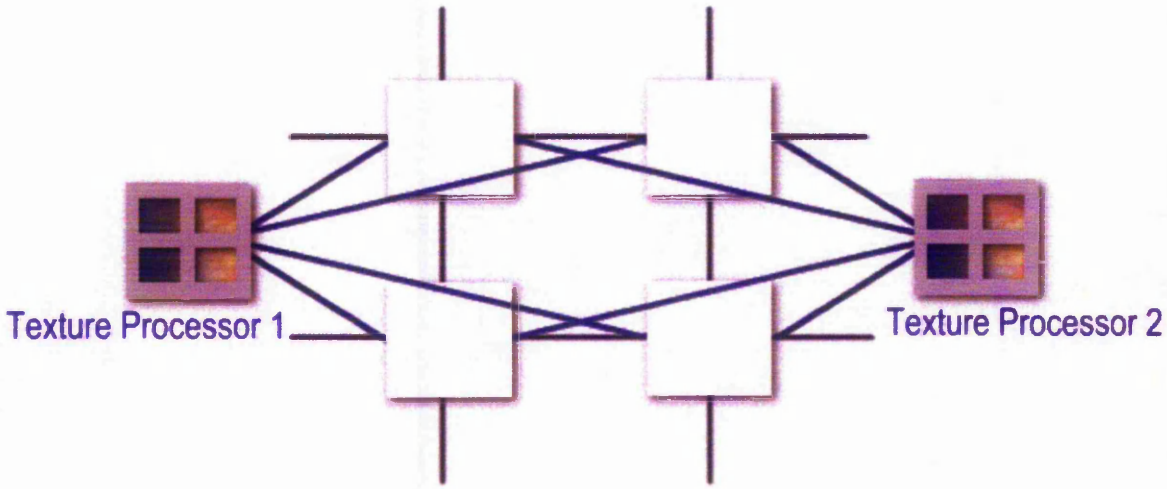


Figure 5.17 **A processor farming strategy dictates that all cells should have access to each texture processor.**

One further disadvantage of this strategy is the increase in the complexity of the communications network required. Such complexity will restrict future re-configuration requirements and as a result will restrict the overall flexibility of the architecture. It may also be argued that performance gains offered by texture processor farming will be somewhat hindered due to access conflicts. This will certainly occur if there are an insufficient number of texture processors in relation to the number of cells in the array.

5.9 Implications of Experimental Results

The simulation results of the cellular array highlight several properties:

1. The benefits of polygon caching are most evident when large datasets are used. Polygon caching also minimises the dependency on the actual distribution of datasets without incurring any performance penalty.

2. The cellular array can employ a wide range of real-time graphics algorithms. Since each general-purpose processor would carry the complete set of algorithms, any change to the system is simply a straightforward coding, recompilation and uploading operation with no consequence to the overall hardware of the system. This 'one suit fits all' strategy means that the inter-processor aspect of the system is hidden from the point of view of the programmer.

3. Techniques can be employed so that the cellular array shows predictable performance scalability as new processors are added. An increase in rendering performance is possible simply by increasing the number of processors in the horizontal direction when an increase in resolution, depth complexity or rendering sophistication is required. Conversely, a greater capability in terms of the number of objects rendered can be achieved by increasing the number of processors in the vertical direction.

4. It is possible that the clock synchronisation required to composite multiple pixel contributions may affect the performance estimates. The multiplexing technology required to re-combine the final image is dependant on the output display technology required and must be able to deal with pixel fragment contributions from multiple cells dedicated to particular regions of the screen. Such device dependency is the only factor influencing the flexibility of the architecture. Composition of information should therefore occur at the last possible moment (i.e. rudimentary pixel data) so that different compositing schemes can be used without influencing the graphics algorithms used in the cellular array. The bus to the multiplexer must therefore run at a sufficiently high speed in order to deal with this, as well as allowing for a variable number of processor banks. This would cater for pixel data to be pipelined before

composition occurs.

If the stream of assembled pixels (RGB values) emerging from the multiplexer are stored in a double-buffered frame buffer, the display can be refreshed at any rate desired so that the responsibility of output synchronisation can be externalised from the array. Furthermore, the addition of processors to the array would require a code recompilation with the provision of an interface to the multiplexer.

5. Parallel TPM for high quality anti-aliased results can best be achieved by parallelising on a per-cell basis as opposed texture-processor farming. The farming scheme is considered too complex in terms of networking if a *re-configurable* system is required. Furthermore, the gains in performance of cubic array are identical to the simulation results for a single processor and therefore provide a much more consistent boost in performance when using TPM.

5.10 Conclusion

Results generated by simulation experiments show that the cellular array provides a new flexible and scalable means to generate high performance 3D graphics. The most prominent factor as to the viability of a cellular array implementation is the cost of localised storage for each processor. The overall requirements are considerably high when compared with other parallel graphics schemes, with 32-64MB DRAM per cell for execution code, depth and pixel buffers, geometric data and textures (pixel-buffer sizes depend on the number of cells per bank). Fortunately, the current and projected costs for high-speed memory are very low. This has already encouraged many hardware vendors to include significant amounts of storage in their graphics technology at high-street prices (particularly for texture storage). Furthermore, the burden of polygon distribution decisions that need to be made to maintain a balanced loading is virtually completely removed from the host processor. This is achieved by localised polygon caching to help overcome distribution dependency.

CHAPTER 6
Conclusions and Future Work

6.1 Conclusions

It can be concluded from the results of this thesis that high quality real-time graphics can be achieved in a flexible and scalable way. From a multi-processing perspective, the cellular array presented provides a long-term solution in a domain that has historically relied on short-term compromises to generate hard-wired results. From a quality perspective, Texture Potential mapping and Texture Potential MIP mapping provide a means to heightened realism in interactive graphics systems at a realistic computational cost.

Several new contributions to graphics research have been presented in this thesis. The most significant of these are summarised below.

- **Texture Potential Mapping**
  This texturing algorithm produces excellent anti-aliased results and is simple to implement in hardware. The algorithm significantly reduces the amount of blurring that is evident in many other texture anti-aliasing algorithms and produces output at a realistic level of performance.

- **Texture Potential MIP mapping**
  This algorithm allows much better control over the balance between quality and performance. By varying this balance, so that the output of this algorithm co-coincides with the qualitative results of other published texturing algorithms it is possible to establish accurate comparisons with respect to performance. Analysis of these results show that Texture Potential MIP mapping is more efficient than other high quality methods.

- **Cellular Array Architecture**

Simulations of the cellular array have shown that, by fusing object space and image space parallelism together, we simplify inter-processor communications and provide a much more flexible and general-purpose architecture.
Solutions to processor network reconfiguration have been provided along with a means to integrate pixel fragment composition schemes for continuous anti-aliasing.

Results also show that the scene dependency inherent in the cellular array can be greatly reduced by applying polygon caching algorithms without placing any burden on the host processor.

- **Parallel Texture Mapping**
  Texture Potential mapping can be implemented in hardware and parallelised by distributing regions of large textures amongst several texture processors. Simulation results of this scheme have shown that high performance gains are possible. A realistic means of integrating this scheme into the cellular array have also been provided.

6.2 Future Work

The following issues separate from the main objectives presented in this thesis are suggested for further study:

Warnock's algorithm, presented in chapter two was stated as offering an interesting extension to continuous anti-aliasing. However, practical applications of this algorithm for pixel fragment composition were not taken far due to the performance costs incurred when compared with other methods. However, the approach itself is quite elegant from an algorithmic perspective. The author therefore offers this as possible future research topic.

Since only the *behavioural* properties of cellular array processing have been studied in order to prove the validity of the architecture, it is hoped that the cellular array will be

analysed further from the perspective of a hardware implementation using up-and-coming technology. The algorithms used to simulate the parallel environment are quite naive in some respects and certainly require further refinement in order to map to a physical hardware environment. Ultimately, a *real* multi-processor prototyping environment would provide the best results and this is recommended as a further course of action.

One suggested enhancement to polygon distribution in the cellular array is to give the host processor greater decision-making powers as to which banks polygon data should be allocated. This has been avoided at all costs during the research to ensure that the host processor does not become a performance bottleneck in a real system. However, a powerful host processor may provide enough idle cycles to be able to 'squeeze-in' additional allocation intelligence. Therefore, it is suggested that this be looked into at a later phase during the hardware development cycle.

The inclusion of parallel TPM into the architecture requires a better estimate on the overall increase in performance. The results depend very much on the texture parameterisation scheme employed and the size of the textures used. A more in-depth study of this is required in future studies.

REFERENCES

[3DLA2000] 3Dlabs inc., "Wildcat 3D Graphics Technology: The Architectural Foundation for 100 ProCDRS performance", http://www.intense3d.com/parascal.html

[ABRA85] Abram, G., L. Westover, and T Whitted, "Efficient Alias Free Rendering Using Bit-Masks and Lookup Tables", SIGGRAPH 85, July, 22-26.

[AKEL88] Akeley, K. adm T. Jermoluk, "High Performance Polygon Rendering", Computer Graphics (Proceedings of SIGGRAPH 88) vol 22, 239-246

[AKEL89] Akeley, K., "The Silicon Graphics 4D/240GTX Superworkstation", IEEE Computer Graphics and Applications, vol 9, July 1989, 71-83.

[APGA88] Apgar, B., B. Bersack, and A. Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS10000", SIGGRAPH 88, 255-262.

[BRES65] Bresenham, J.E., "Algorithm for Computer Control of a Digital plotter"" IBM Systems, 4(1); 25-30, 1965

[CANT96] Cant, R.J., "Optimising Display Update Rates in Virtual Reality Systems", Simulation In Industry ESS96 Conference Proceedings, Italy, October 1996, 24-28

[CANT2000] Cant R.J., Shrubsole, P., "Texture Potential MIP Mapping, A New High Quality Texture Antialiasing Algorithm", ACM Transactions on Graphics Volume 19 No. 3 July 2000 (to appear in October)

[CARP84] Carpenter, L., "The A Buffer, an Anti-aliased Hidden Surface Method", SIGGRAPH 84, vol 18, 103-108.

[CATM74] Catmull, E.E., "A Subdivision Algorithm for Computer Display of Curved Surfaces", PhD Dissertation, university of Utah, Dec 1974

[CROW84] Crow, F., "Summed Area Tables for Texture Mapping", Computer Graphics, vol 18, July 84.

[DUDG91] Dudgeon, A., "Algorithms for Texture Mapping", In Proceedings of the 23rd South Eastern Symposium on Systems Theory, March 1991, 613-617

[EVAN91] Evans and Sutherland Computer Corporation, ESIG4000 Technical Overview, 600 Komas Drive, Salt Lake City

[EYLE88] Eyles, J., J. Austin, H. Fuchs, T. Greer, J. Poulton, "Pixel Planes 4: A Summary", Advances in Computer Graphics Hardware II, Eurographics Seminar, 88, 183-208

[FEIB80] Feibish, A., M. Levoy, L. Cook, "Synthetic Texturing using Digital Filters", Computer Graphics vol 14, July 80

[FOLE90] Foley & van Dam [et al], "Computer Graphics : principles and practice", 2nd edn. Addison-Wesley, 1990, 874-900

[FUCH81] Fuchs, H., J. Poulton, "Pixel-Planes: A VSLI-Oriented Design for a Raster Graphics Engine", VLSI Design 2(3), Q3 81, 20-28.

[FUCH89] Fuchs, H., J. Poulton, J. Eyles, T.Greer, J. Goldfeather, D.Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Isreal, "Pixel Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories", SIGGRAPH 89, 79-88

[GREE86] Greenberg, D.P., M.F. Cohen, and K.E. Torrance, "Radiosity: A Method for Computing Global Illumination", The Virtual Computer 2, 1986, 291-297

[GLAS86] Glassner, A., "Adaptive Precision in Texture Mapping", Computer Graphics, vol 20, November 86.

[HAEB90] Haeberli, P. and K. Akeley, "The Accumulation Buffer: Hardware Support for High Quality Rendering", Computer Graphics (Proceedings of SIGGRAPH 90) vol24, 309-318

[HECK88] Heckbert, P., Survey of Texture Mapping, Computer Graphics: Image Synthesis, Computer Society Press, 1988, 321-322

[MITC87] Mitchell, D.P., "Generating Anti-aliased Images at Low Sampling Densities", SIGGRAPH 87, 65-72

[MOLN90] Molnar, Steven and Henry Fuchs, "Advanced Raster Graphics Architecture", Chapter 18 in Computer Graphics: Principles and Practice by James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes, addison-Wesley, New York, 1990, 855-922.

[MOLN92] Molnar Steven., John Eyles, John Poulton, "PixelFlow: High Speed Rendering Using Image Composition", Computer Graphics 26, July 92, 231-240

[NVID2000] Nvidia Corporation, Transform and Lighting Technical Brief, http://www.nvidia.com/Products/GeForce2ultra.nsf/second.html, August 2000

[OPPE75] Oppenheim, A.V. and Shafer, R.W., Digital Signal Processing, Prentice Hall. Englewood Cliffs NJ, 1975.

[POTM83] Potmesil, M., and I. Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model", ACM TOG, April 1982, 85-108

[ROSE73] Rose, A., Vision: Human and Electronic, Plenum Press, New York, 1973

[SCHA80] Schachter, B., "Long Crested Wave Models", Computer Graphics and Image Processing, 1980, 12:187-201

[SCHA83] Schacter, B., Computer Image Generation, John Wiley & Sons, Inc, New York, 1983

[SCHI93] Schilling, A., "EXACT: Algorithm and Hardware Architecture for an Improved A Buffer", SIGGRAPH Annual Conference Proceedings, August 1993, 85-92.

[SCHI96] Schilling, A., "Texram: A SmartMemory for Texturing", IEEE Computer Graphics and Applications, May 1996, 32-41.

[SGI2000] SGI inc, SGI Onyx 3000 datasheet, Corporate Office, Mountain View U.S., August 2000

[SHRU96] Shrubsole, P., R.J. Cant, "Proposal for Distributed Texture Mapping", Simulation In Industry ESS96 Conference Proceedings, Italy, October 1996, 29-33.

[SHRU97] Shrubsole, P., Cant R.J., "Texture Potential Mapping: A Way to Provide Anti-aliased texture without Blurring", Visualisation and Modelling , Academic Press, December 1997, 223-240

[SHRU98] Shrubsole, P., "Cubic Cellular Graphics Processing: A Simulation", Simulation Technology: Science and Art, 10th European Simulation Symposium Proceedings, October 1998, 666-670

[SUTH63] Sutherland, I.E., "Sketchpad: A Man-Machine Graphical Communication System", SJCC, Spartan Books, Baltimore MD 1963.

[SZAB83] Szabo, N., "Digital Image Anomolies: Static and Dynamic", in Computer Image Generation, Bruce J., Schachter, Ed., John Wiley & Sons, New York, 1983, 125-135.

[WARN69] Warnock, J., "A Hidden Surface Algorithm for Computer Generated Half-Tone Pictures", Technical Report TR 4-15, NTIS AD-753 671, Computer Science Dept, University of Utah, June 1969.

[WATT98] Watt A. and Watt M., "Advanced Animation and Rendering Techniques", ACM Press, New York, 125-127.

[WHIT80] Whitted, Turner., "An Improved Illumination Model for Shaded Display". ACM June 1980, 343-349

[WILL83] Williams, L., "Pyramidal Parametrics", Computer Graphics vol 17, July 1983

Publications by Author

Proposal for Distributed Texture Mapping, Simulation In Industry ESS96 Conference Proceedings, Italy, October 1996, 29-33.

Texture Potential Mapping: A Way to Provide Anti-aliased texture without Blurring, Visualisation and Modelling, Academic Press, December 1997, 223-240

Cubic Cellular Graphics Processing: A Simulation, Simulation Technology, 10th European Simulation Symposium Proceedings, October 1998, 666-670

Texture Potential MIP Mapping, A New High Quality Texture Antialiasing Algorithm, R. J. Cant and P. A. Shrubsole, ACM Transactions on graphics, July 2000 (to be printed in October)