Appendix A

Other considerations and approaches

This appendix starts by discussing a number of different approaches applicable to individual stages in the rendering pipeline. This is followed by a description of various approaches to parallelism in graphics rendering systems and then by a description of completely different methods of rendering. The appendix ends by describing a number of errors and inaccuracies encountered in rendering and some of the ways that these may be avoided.

A.1 Model format

Planar polygons (and occasionally curved surface descriptions) have been the main focus of attention of this thesis, however a number of other representations exist which may be more suitable for some applications.

A.1.1 Particle systems

The appearance and behaviour of fire, smoke and liquids are often hard to define, the particle systems approach models these types of objects as a large collection of *particles* each with its own behaviour. Each particle typically has a number of characteristics such as its colour, its position in space and its rate of movement; the *behaviour* of each particle updates these characteristics based on a number of rules such as gravity and air flow simulations. Particles are then rendered as simple points or small spheres; it is a collection of a large number of particles which produces the appearance and behaviour of a complex object such as a smoke cloud. The rendering characteristics of this type of model are quite different from a conventional polygonal model; in general there are many more primitives but each one is a lot simpler. Particle systems techniques can be used to enhance a polygonal scene.

A.1.2 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a representation suitable for solid models more commonly used in mechanical design systems. Using a conventional hierarchical polygonal model it is difficult to represent a block with a hole drilled in it without resorting to a unique polygonal description. In a CSG representation the same block would be represented by a block, a cylinder and a logical operator combining the two. CSG models consists of trees whose leaf nodes contain objects (such as cylinders) and whose internal nodes are Boolean set operations which combine subtrees (such as 'union' and 'intersection'). CSG models can be rendered using a modified depth buffer algorithm, such as that described in **[GoHF86]**.

In general CSG representations are convenient for representing objects which are to be produced by machining, since common mechanical operations such as drilling can easily be represented as a Boolean expression.

A.1.3 Height fields

One way of representing a terrain model is as a *height field* which is an array of heights usually placed on a regularly spaced grid. This model is efficient to store and transform and can easily be converted into a polygonal mesh for display. In addition it is sometimes possible to optimise rendering algorithms to address the specialist needs of representations such as the height field.

A.1.4 Surfaces of revolution

The *surfaces of revolution* representation can be used to represent a restricted set of objects which are cylindrical; e.g. wine glasses and spheres. The model is stored as an outline profile which is then swept around an axis to create a model of the shape; Figure A.1 shows a simple outline and the resulting shape[1]. The advantage of this type of model is that it is easy to render at various levels of detail **[Crow82]** and it uses very little storage in its most basic form; the example shown in the figure needs to store only nine points.

Figure A.1: A Surface of revolution

A.2 Hidden surface removal

This thesis has concentrated on the Z buffer hidden surface removal algorithm. This algorithm is used in nearly all commercially available systems because of its simplicity and ease of implementation in hardware. There are however many other hidden surface removal algorithms which have been proposed and have been used in rendering systems but which have now fallen out of use. The Z buffer method was originally regarded by many as impractical due to the large amount of memory required, however as memory prices have fallen, the cost of the Z buffer itself has become less of a problem. Although Z buffering is simple it does have a number of problems. The first is that it can suffer

1. Created using the Silicon Graphics demonstration program *revo*.

from the limited accuracy of the values stored in the Z buffer; this problem is described in detail in section **A.8.2**. Most implementations of Z buffering create the need for a costly read-modify-write cycle which may take twice as long as a single write; one solution to this problem is FBRAM described in section **6.3.2**. Since the Z value has a larger range than the colour values, the interpolation of the Z value can dominate the time required to produce a new pixel. Finally as mentioned in **Chapter 5** , the Z buffer technique renders all primitives, even those that are hidden from view; this is often considered wasteful.

This section describes a number of algorithms following the general order of the descriptions given in Chapter 15 of **[Fole90]**. Further details of these algorithms (and others) can be found in **[Fole90]** and **[Suth74]**.

A.2.1 List-priority algorithms

List-priority algorithms reorder the objects from which the scene is built to ensure that objects closer to the viewer are drawn later and thus over objects which they obscure. To accommodate intersecting and cyclically overlapping polygons it may be necessary to subdivide some of the primitives or objects during the reordering process. The rasterisation stage of these algorithms does not suffer from many of the problems of the Z buffer algorithm although in general they still render primitives which are hidden from view only to overwrite them later.

The simplest list-priority algorithm (created by Newell, Newell and Sancha) involves sorting the primitives in a back-to-front order and splitting primitives to resolve problems caused by intersecting or cyclically overlapping primitives. The resulting ordered list or primitives is then rendered in back-to-front order[1]. A full sort of primitives is

1. A simplified version of this algorithm (based on application generated priority values) is known as the painter's algorithm since it is similar to the way in which a painter may start at the furthest object and work forward to the closest ones

expensive and typically has a time complexity no better than O(n log n) for the average case. The cost of this method may be reduced by only sorting primitives which overlap in screen space and by taking advantage of the similarity in depth ordering from frame-to-frame. A In addition to the complexity of the sorting process, the need to check for (and eliminate) intersecting and cyclically overlapping primitives is also responsible for a large amount of computation. Since each of these processes must be carried out on every frame before rasterisation can begin, this method introduces a large latency making it unsuitable for VR applications.

A more complex list-priority algorithm is based around the '*Binary-Space Partitioning tree*' (BSP tree) **[FuKN80]**. The BSP tree represents a partitioning of space which can be used for a number of purposes, including hidden surface removal. To build a BSP tree one polygon is selected from the model and placed in the root node of the tree. Polygons behind the root polygon (as defined by its surface normal) are placed in the right hand subtree and those in front of the polygon are placed in the left hand subtree. This procedure is repeated to build each subtree. Displaying the model stored in the tree is accomplished by recursively rendering the subtree furthest from the viewer, then the polygon at the root of the tree and then the nearer subtree. The advantage of the BSP method is that most of the complex calculation is performed outside the time critical display process. The BSP algorithm suffers from one main disadvantage; it is difficult to modify and thus unsuitable for scenes which change frequently. The BSP tree can also be used to perform view frustum culling.

A.2.2 Scan-line algorithms

In general the Z buffer algorithm performs all operations on one particular primitive (wherever that primitive may occur on the screen) before moving on to the next primitive. In contrast to that approach the scan-line algorithms perform all operations on a particular scan-line before moving on to the next one.

Although it would be possible at each scan-line to test every primitive for intersection with that scan-line it would be inefficient since only a small proportion of primitives intersect with each scan-line. A common solution to this problem is to transform all primitives and to bucket sort their edges based on their smallest Y coordinate. The result of this bucket sort is the *edge table*, which is a series of lists (one per scan-line) which contain all edges for which this is the first scan-line which the edge intersects. For each edge this table contains information concerning the initial X coordinate of the edge, the slope, the Y coordinate of the other end of the edge and a pointer into a *polygon table* for the polygon to which the edge belongs[1]. The polygon table contains information including the plane equation of the polygon and shading information. Depending on the particular type of scan-line algorithm used, extra information may be held in both the edge and polygon tables. Rather than traversing the entire edge table at each scan-line, a third table, the *active edge table* (AET), is used which contains all edges which interact with the current scan-line. All parameters in the AET are updated as the algorithm moves to the next scan-line. The AET is sorted in order of the current X intercepts of the edges. In general scan-line algorithms make great use of scan-line to scan-line coherence both in the determination of edge/scan-line intercept points, maintaining the order of the AET sorted on X, the determination of shading information and hidden surface removal. The main difference between the various scan-line algorithms is the way in which the AET is rasterised at each scan-line. In general all of the algorithms suffer from a large latency due to the need to transform (and bucket sort) all primitives before rasterisation can begin. The edge table can be quite large, and usually requires double buffering to allow the transformation and sorting of one frame while the next is being rasterised. One major advantage of scan-line algorithms is that they lend themselves to parallelisation due to the fact that each scan-line is largely independent of the others.

1. In some algorithms, such as the spanning scan-line algorithm used in **[Vial93]**, each entry in the edge table contains two polygon pointers one for the polygon on either side of the edge. This allows shared edges to be stored only once.

A.2.2.1 The scan-line Z buffer

One of the criticisms of the Z buffer is the need for a large amount of memory to hold the Z buffer during rendering; the scan-line Z buffer algorithm removes this problem by holding just one scan-line of Z buffer[1].

At the start of each scan-line the Z buffer is cleared to the furthest Z value. Pairs of edges from the same polygon are then grouped together to form spans which are then rendered in to the scan-line Z buffer in a similar manner to a conventional rendering algorithm. If the only aim of using the scan-line Z buffer is to reduce the amount of memory used, the colour information can be stored in the main frame buffer, however it is also possible to reduce the bandwidth to the main frame buffer. To do this a scan-line colour buffer is also included and at the end of rendering a scan-line, its contents are copied to the main frame buffer. In this way the bandwidth to the main frame buffer is completely predictable and (unlike the conventional Z buffer algorithm) is not related to the depth complexity of the scene. This method also has the advantage that the scan-line buffers are small and can thus be made out of faster (more expensive) memory thus providing a very high bandwidth between the span fillers and the memory. It may be beneficial to include pointers in the edge table and active edge table to ease the pairing of edges from the same polygon.

A.2.2.2 The Spanning Scan-line Algorithm

This algorithm subdivides each scan-line into *spans*, which are areas where the same polygon is visible in the entire area. It performs this subdivision by traversing the AET from left to right, adding or removing polygons from an *Active-Polygon list* as each edge is encountered. This active-polygon list is sorted on depth, so that the first entry on the list is the closest to the viewer. As each edge is encountered a span can be filled from the

1. For highly complex scenes or low resolutions the double buffered edge table may be larger than the Z buffer being replaced.

intersection of the previous edge with the scan-line, to the intersection of the present edge using the material properties of the first polygon on the active-polygon list.

Where polygons intersect it is necessary to introduce extra edges along the lines of intersection to split spans up. Since hidden surface removal is completed *before* rasterisation of spans or shading calculations are performed pixels in spans are only written *once*. One advantage of the Spanning scan-line algorithm is that accurate (none refractive) transparency can easily be accommodated at little extra cost **[Vial93]**. Using an algorithm of this type it would be possible to store the spans rather than individual pixels in the frame buffer and perform interpolation along spans as the display is refreshed. For low complexity scenes this technique is possibly advantageous since it is likely to use less display memory; however for scenes with a greater level of complexity, where the spans tend to be much shorter, it is possible that the amount of storage required for spans could be greater than that required for a full pixel level frame buffer. Addressing a span buffer of this type is difficult since each scan-line requires a different amount of storage. One other problem with this type of frame buffer is that while systems using a conventional pixel based frame buffer and systems using a span buffer need to interpolate the same number of pixel values over an entire frame, the system using the span based frame buffer must be able to produce pixels at a constant rate, even on complex lines with many short spans.

Watt summarises the relative performance benefits of the spanning scan-line algorithm when compared with the Z buffer algorithm as follows:

> *"It is generally claimed that spanning algorithms are more efficient than Z-buffer based algorithms, except for very large numbers of polygons (**Foley and van Dam, 1982; Sutherland et. al., 1974**). However, since extremely complex scenes are now becoming the norm, it is becoming clear that, overall, the Z buffer is more efficient, unless a very complex shading function is being used."* **[Watt89]**, p.107.

A.2.3 Ray casting

To construct an image using ray casting a ray is 'shot' from the viewpoint through the centre of each pixel on the display and into the model. For each ray an intersection test is carried out between every object in the scene and that ray. If no objects intersect the ray then the pixel may be coloured in the background colour. If the ray intersects more than one object then it is necessary to find the intersection closest to the viewer. To shade the pixel through which this ray passes it is necessary to calculate the normal at the point of the ray/object intersection and then use a conventional lighting model, such as the one described in section **3.8**.

A highly versatile extension to ray casting, *recursive ray tracing*, is described in section **A.7.1**.

A.3 Interpolation techniques

In the conventional rendering pipeline it is necessary to interpolate values for span start and end positions, colour and depth positions along edges and colour and depth position alone during span filling. The simplest method of implementing this is by the *Digital Differential Analyser* (DDA); given a start and end position, x_s and x_e, and a value at the start and end positions, i_s and i_e, the DDA algorithm works out a delta value representing the difference between each step. This delta is then added to the current value to produce each step.

There are two main problems with this approach. The first is that the initial calculations for the delta value require a division which is costly. The second is that a high degree of accuracy and a large range are needed. To accommodate such a large range of values floating point representations are often used; many systems use fixed point values during the iteration loop and floating point for the initialisation. These fixed point values often require a sizeable fractional section to retain accuracy over a large span; interpolators for these large fixed point values are slow due to the potentially long carry propagation; faster implementations are possible but require large amounts of logic and are expensive.

One solution to both of these problems is to separate the calculation of the fractional and integer sections of the value being interpolated. A large addition can be split into two separate shorter additions as long as any carry generated by the addition representing the less significant section is passed onto the more significant section. When interpolating edge coordinates it is often possible to guarantee that the delta value is less than one and it is thus possible to eliminate the addition for the integer part.

It is possible to rearrange this algorithm to avoid the use of the initial division and to remove the need for any non-integer arithmetic. This then becomes Bresenham's algorithm; as described in **[SwTh86]**. This is only suitable for interpolation of edge coordinates since it requires that the delta value is guaranteed to be less than one; Swanson and Thayer in **[SwTh86]** extend Bresenham's algorithm to perform interpolation of colour and depth values. Their algorithm performs all interpolation using integer arithmetic, thus allowing small registers and ALUs to be used in the time critical span fill stages of the renderer. In addition Swanson and Thayer's algorithm replaces the expensive floating point division needed by conventional interpolators by an integer division and remainder function; the remainder is effectively available without extra expense after the integer division.

A.4 Deferred shading

One of the problems with a Z buffer based architecture is that many of the pixels that are rasterised and shaded are not actually visible. Besides the wasted frame buffer bandwidth, another effect is that the computation involved in complex algorithms (such as Phong shading and some texturing algorithms) is also wasted. One solution to this problem is *Deferred shading*, where the shading process operates after hidden surface removal and rasterisation is complete. Typically systems based on this concept produce interpolated values, representing such properties as surface normal and position in a texture map, during the rasterisation process in much the same way as conventional rendering systems. These interpolated values are then passed to a shading engine after the hidden surface problem has been solved. Various different systems organise the storage and communication of these intermediate values in different ways. PixelFlow **[MoEP92]** described in section **6.1.7.7** is specifically designed to operate using deferred shading. One of the main advantages of deferred shading is that it sets an upper limit on the amount of work required to shade the scene, which is purely related to the resolution and refresh rate of the display

A.5 Pineda's Algorithm

One way of defining a convex polygon is as the intersection of a number of *half-spaces* defined by the polygons edges. An edge 'i(X,Y)' is represented by an edge function which is positive if the point (X,Y) is in the half space which includes the polygon, negative if in the other half space, and zero if the point is on the edge. It is possible to use this definition to rasterise the polygon by evaluating the edge functions at every pixel and illuminating all pixels where all edge functions are positive. [1]Evaluating this function at every pixel is very expensive and impractical in most architectures; the exception to this is the Pixel-Planes architecture described in section **6.1.7** which evaluates edge functions at every pixel simultaneously.

Pineda's algorithm **[Pine88]** provides an alternative method of evaluating these functions. Pineda notes that the edge functions can be computed iteratively by a simple addition or subtraction from neighbouring pixels:

1. Special cases must be defined to take account of pixels which are on edges.

$$
\begin{aligned}
E_i(X+1, Y) &= E_i(X, Y) + dY_i \\
E_i(X-1, Y) &= E_i(X, Y) - dY_i \\
E_i(X, Y+1) &= E_i(X, Y) - dX_i \\
E_i(X, Y-1) &= E_i(X, Y) + dXi
\end{aligned}
\qquad \textbf{EQ A.1}
$$

To perform rasterisation it is therefore necessary to select a starting point and then traverse the space covered by the triangle one pixel at a time. The most complex problem in Pineda's algorithm is the selection of an efficient traversal algorithm. Various methods are describe in **[Pine88]**; the simplest being to visit every pixel in the bounding box of the polygon (Figure A.2). In many cases this will visit many pixels which are not

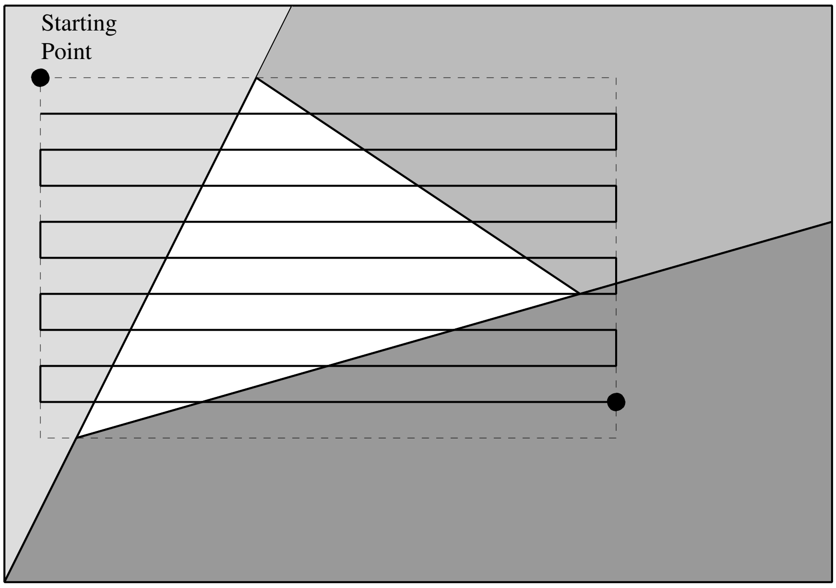

Figure A.2: A simple traversal method

contained within the polygon and is thus inefficient. Another method described by Pineda is shown in Figure A.3. The method starts at a pixel near the top point of the polygon. At each stage the algorithm stores the current point and then scans left and right, until it leaves the polygon. At that time the algorithm restores its states and moves down a line. At the start of each line it is necessary to ensure that the current point is in the polygon and if not, search until the edge of the polygon is found. A hardware implementation of Pineda's algorithm contains a number of interpolators, each capable of adding or

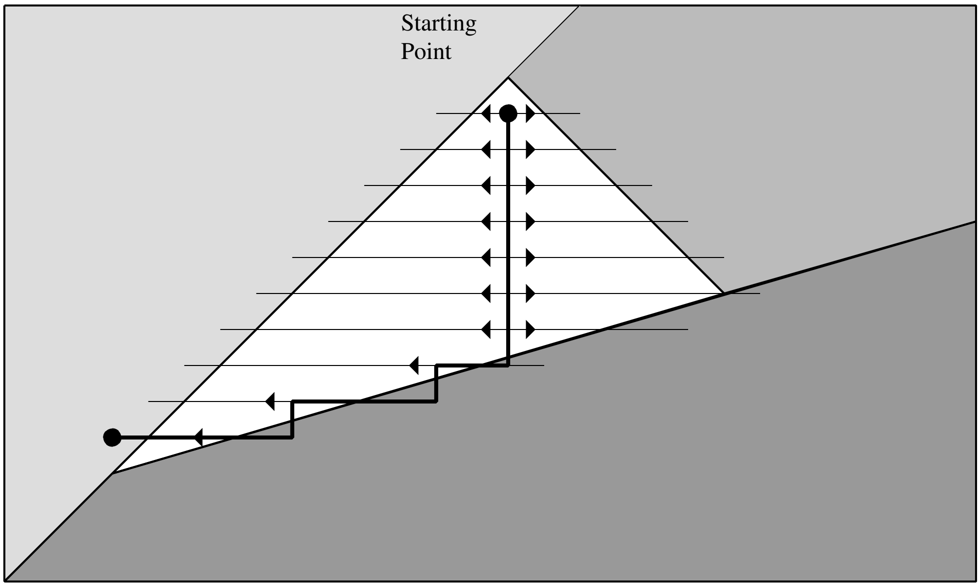

Figure A.3: A more advanced traversal algorithm

subtracting either the X or Y step for each parameter or edge value. One of these interpolators is needed for each edge function. A state machine is used to perform the traversal algorithm and detect the edges of the polygon.

So far this description has only covered rasterisation and has not included shading or hidden surface removal. The colour and depth information at any pixel on a planar polygon can be interpolated in a similar fashion to the edge functions. Thus rasterisation, shading and hidden surface removal calculations can be performed by seven interpolators operating in parallel (three for the edge functions, three for colour, and one for depth information). The width of the data path in each interpolator is generally different, to reflect the range of the values being interpolated.

Pineda's algorithm offers a number of advantages over the conventional rendering algorithm:

- Simplicity

  The whole process is simpler than the conventional rasterisation process. The conventional process requires the polygon to be split into axis parallel trapezia, spans generated and then each of those spans filled; often requiring a number of stages in a pipeline. Pineda's algorithm just requires one simple set of hardware which can be integrated into one chip.

- Simple parallelisation

  EQ A.1 shows how to produce the edge function value for the neighbouring pixel. It is simple to extend this to step two pixels at a time:

  $$E_i(X+2, Y) = E_i(X, Y) + 2dY_i \qquad \textbf{EQ A.2}$$

  Thus by having two interpolators (whose initial values are different) two pixels may be produced simultaneously. This process may be extended to interpolate whole blocks of pixels at once and thus take advantage of certain interleaved memory architectures.

- Integral clipping

  A window (or other clipping area) can be represented as a number of edge functions and interpolated in the same manner as polygon edge functions. It is then simple to modify the traversal algorithm to stop at clipping edges in the same way that it does for polygonal edges.

- Sub-pixel vertex accuracy

  In conventional rendering systems it is not uncommon to round the X and Y coordinates of a vertex to integer values before commencing rasterisation. This is inherently inaccurate and leads to gaps or overlaps between primitives. Pineda shows that as long as the setup calculations for his algorithm are performed accurately and interpolation is performed in fixed point arithmetic with suitable precision, these inaccuracies will not occur.

- Antialiasing

  The edge functions can be processed to provide a measure of the distance of a pixel centre to the edge; this value can then be used to generate information for antialiasing algorithms.

One disadvantage of Pineda's algorithm is that in most traversal methods, pixels are generated in both left to right and right to left order. This complicates frame buffer accesses especially if old Z values are recovered from the serial access port of VRAM as described in section **3.14.3**.

The second traversal scheme described above effectively splits the filling of the pixels on a scan line into two smaller sections (one right of centre and one left of centre); this reduces the speed advantages gained by filling consecutive pixels.

A.6 Parallelisation

Rendering is no different from any other field of computing in that the problems which people are trying to solve are often beyond the current capabilities of a single processor. The obvious (and normally most efficient) solution is to build a faster processor. Unfortunately the current level of technology normally presents a limit to the capabilities of a single processing device. Another solution is to use more than one processing device. In general two different approaches exist to parallelisation. The first is specialisation, where a number of different types of processing stage are used, the second is duplication where a number of identical processing stages are used. The following sections describe how these approaches are applied to rendering systems.

A.6.1 Pipeline parallelism

Pipeline parallelism is a form of specialisation and has been used in the rendering system described in **Chapter 3**. This method is often compared to a production line approach where the task is split into a number of simpler subtasks which can be implemented more efficiently. Pipelining increases throughput, since the total processing power in the pipeline is typically greater than that of a uniprocessor system, and each stage is optimised for its role in the rendering task. Two problems exist - latency and load balancing. Pipelined systems often incur latency; the amount of latency incurred depends on how finely the task is subdivided and the organisation of the data passed between stages. Figure A.4 shows the timing behaviour of an unpipelined system based

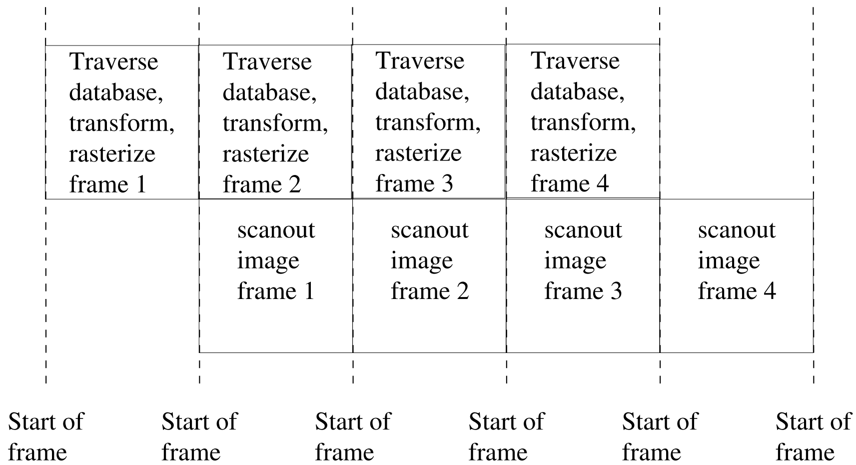

Figure A.4: Activity in an unpipelined rendering system

on one processor. The latency from the start of the rendering process (traversal of the database), to the start of scan-out, is one frame. One organisation of a rendering pipeline stage could process all the operations for one frame before passing the results to the next stage. The flow of operations for a system with this type of stages is depicted in Figure A.5. The latency of this system is three frames. It is possible to render a frame with only one frame of latency with such a system by using each step for only a third of a frame as shown in Figure A.6. In general this system is inefficient since most of the hardware lies idle during a large part of the frame.

The system described in **Chapter 3** does not suffer from this problem since in an immediate mode configuration the data flowing in the pipeline between the traversal, transform and rasterisation stages are individual primitives rather than groups of primitives for whole frames; this is shown in Figure A.7. As soon as the traversal process produces

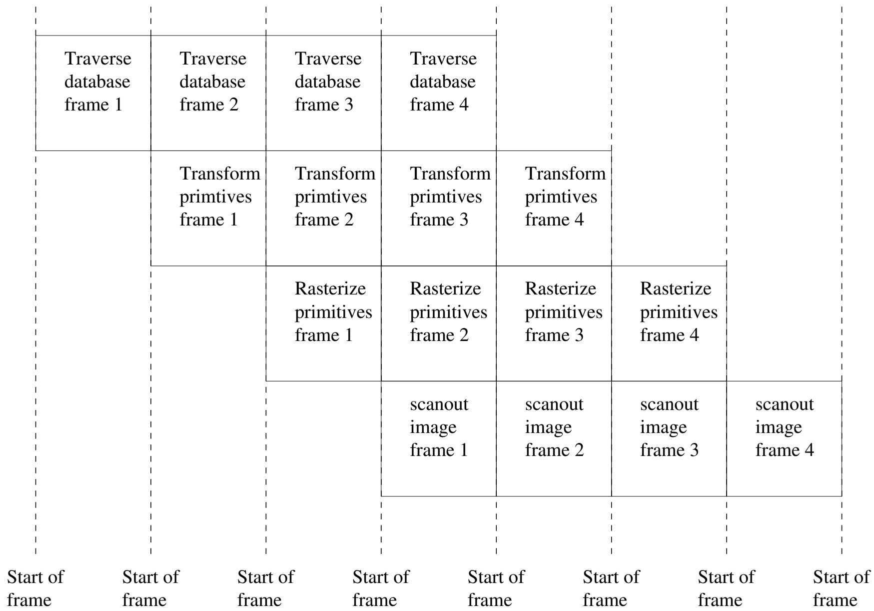

Figure A.5: Activity in a pipelined rendering system

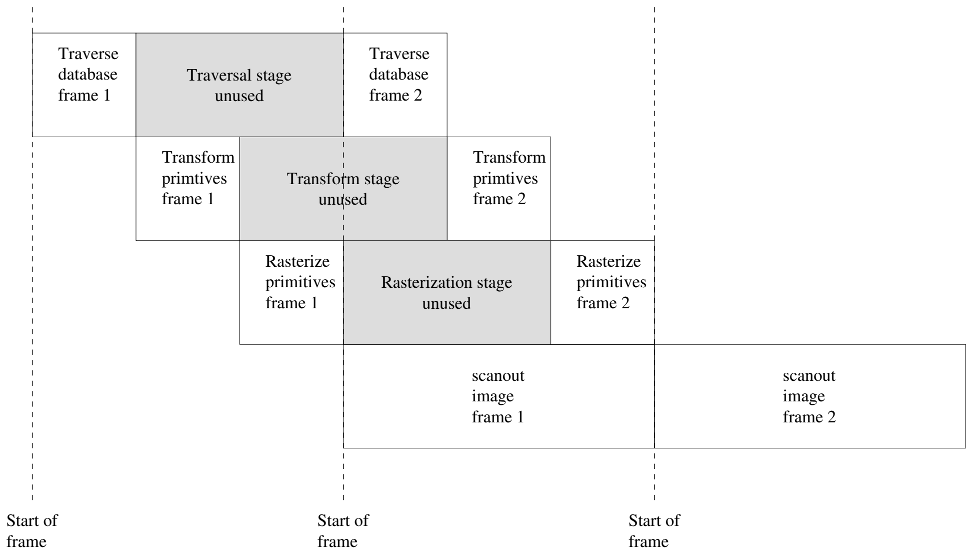

Figure A.6: Pipelining and low latency by low utilisation

the first primitive, the transform process can start to transform it. Similarly, as soon as that primitive has been transformed, the rasterisation process can begin. The latency is therefore just over one frame[1]. Note that in this system, the start of rendering a frame does not necessarily coincide with start of scan-out of a frame.

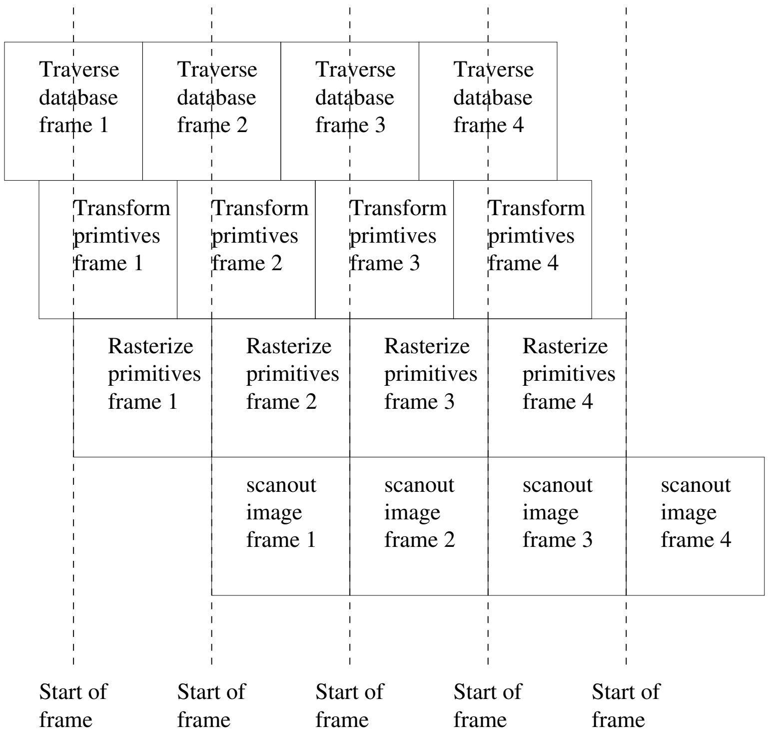

Figure A.7: Overlapping pipeline stages in an immediate mode system

In a retained mode system, extra latency is added due to the fact that all editing of the database must finish before traversal may begin (Figure A.8). The amount of latency now depends on the amount of editing to be done. The systems depicted in all these figures have been systems which use double buffering and thus suffer one extra frame of latency.

One difficulty in pipeline systems is maintaining a *balanced* pipeline. In a scene containing many small primitives, the rasterisation stage may be able to compete its task much faster than the geometry system can supply primitives. Similarly scenes containing many large primitives cause the geometry stage to wait for the rasterisation stage to fin-

1. It is not exactly one frame since the stages do not completely overlap while processing the first and last primitives.

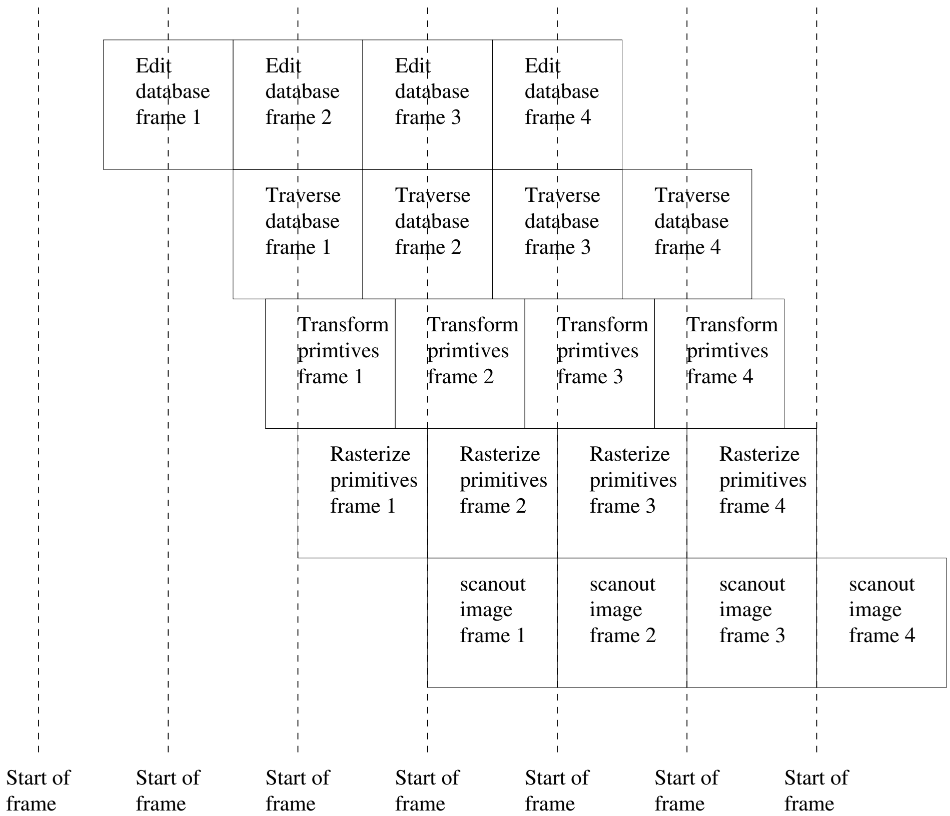

Figure A.8: Pipeline activity in a pipelined retained mode system

ish. This often leads to cases where one or more sections of the pipeline are under utilised. In scenes with a mixture of primitive sizes the problem still exists, since after processing a large primitive, the geometry engine will have to wait for it to be rasterised before transforming the next primitive. The frequency with which this occurs can be reduced by placing buffering (in the form of first-in-first-out (FIFO) buffers) between each stage of the pipeline so that the geometry processing for a number of small primitives can be performed while a large primitive is rendered.

A.6.2 Screen subdivision

A different approach to parallelism using duplication (which is often used in conjunction with pipelining) is to divide the screen up into a number of areas and render each area using separate hardware.

The simplest form of screen subdivision is to split the video memory up into a number of banks and to take pixels out of each bank in turn. This system is called *interleaving*. This increases the bandwidth available to both the display subsystem (allowing higher resolution displays and higher refresh rates) and the rasterisation subsystem (allowing higher span fill rates). The increase in span fill rates comes from the ability to generate pixels for all banks simultaneously; the effectiveness of this depends on the way in which interleaving is performed. Simple systems perform horizontal interleaving; i.e. consecutive pixels on a scan-line are stored in different banks. This allows long spans to be filled very fast. The disadvantage of this approach is that triangles which are tall, but only narrow, are not accelerated. The most serious implication of this for VR applications is that a scene may be rendered at different speeds when rendered from different angles. Many commercial systems (such as the Silicon Graphics VGXT described in section **6.1.1**) interleave both horizontally and vertically having multiple span generators as well as fillers. This produces rectangular blocks of pixels which are filled simultaneously to attain extremely high fill rates which are almost independent of the shape of the triangle.

Another approach to screen subdivision is to split the screen into a number of individual blocks and assign each block to a separate renderer. Typically such systems have less renderers than blocks on the screen and must use each renderer to render a number of blocks. The blocks can be rectangular areas of the screen (as in Pixel-Planes 5 **[FPEG89]**) or scan lines.

Typically, the entire database is traversed and transformed and the resulting primitives are then sorted into bins representing the area of the screen which they occupy (or the first scan-line which they occupy). Each renderer then takes one bin, renders it and writes the result to the frame buffer. After it has finished that bin it takes another until all bins have been processed. This is an example of *dynamically allocating* renderers to areas of the screen. Another approach is *static allocation* where the areas of the screen each renderer will process is determined in advance. Both of these approaches introduce considerable extra latency since the transformation stage can not be overlapped with the rendering stage.

A modification to this scheme is to transform the scene as normal and immediately start sending out primitives to the renderers. Renderers move blocks of image data from the frame buffer, render primitives into that area and then return that area to the frame buffer. In this way each area of image may be processed by a renderer more than once but there is no need to sort the primitives and thus no extra latency is incurred. This process is inefficient since image blocks are moved back and forward to the frame buffer memory; this problem is not quite as serious as it could be for two reasons. First all the primitives for one object tend to be sent down the pipeline at once, these primitives tend to occupy the same areas of the screen and thus these areas are already loaded into the renderers. Secondly the transfer to and from the frame buffer is done without Z comparisons since the depth comparison is carried out in the local memory of each renderer and thus transfers to frame buffer memory are faster than the read-compare-write cycles needed for Z buffered writing.

Systems which divide the screen into scan-lines for rendering operate in a slightly different manner; section **A.2.2** describes a class of algorithms more suited to these systems while section **6.1.6** describes an architecture which uses scan-line parallelism.

One problem with screen subdivision into blocks, is that it is not uncommon for areas of scene complexity to cluster in just a few of these blocks. This causes some renderers to take much longer to render their section than others, and in the case of systems using static allocation of renderers, can cause the rendering times to vary greatly as objects move across different parts of the screen. Another major problem is that it is necessary to store all the primitives in a sorted state between the transformation and the rasterisation stages. This requires very large amounts of memory and also places a limit on the complexity of the scene that can be rendered. One final problem is that the communication of the primitives between the geometry subsystem and the renderers requires a very high bandwidth communication system capable of connecting any geometry engine to any renderer. This communication system is difficult and expensive to build.

A.6.3 Image composition

Image composition is a class of object parallelism which overcomes many of the problems of screen subdivision. Architectures such as PixelFlow (described in section **6.1.7.7**), which are based on Image composition, consist of a number of independent rendering engines. Objects from the model are distributed among the rendering engines. Each engine renders the objects it has been passed independently and the images from all the renderers are combined together later by the process of composition. The point at which objects are distributed to the renderers varies from one system to another. Some systems include a geometry processor on each renderer; thus removing the need for the highspeed communication and routing network between the geometry subsystem and the renderers.

At one extreme of image composition systems there exist architectures which have a processor for each object in the model. One such system is *PROOF* which is described in more detail in section **6.1.5**. At the other extreme lie architectures which hold a complete frame buffer for each renderer. In these systems the renderers are often similar to conventional rendering engine and each one receives a large number of primitives to render. Composition in these architectures is performed by a tree of compositors. In their simplest form a compositor is a comparator and multiplexer; as input it takes two streams of pixels and outputs a stream of pixels containing the closest pixel out of each pair input. Each renderer is connected as a leaf of the tree with the output being from the compositor at the root. There are two main problems with this approach:

- Memory usage

  Each renderer has a complete frame buffer (including depth buffer).

- High speed of compositors

  The compositors have to run at very high speeds; this problem is not particularly serious since each compositor is extremely simple and thus lends itself to a high speed design.

The main advantage of image composition architectures is that they scale very easily and almost without limit.

A.6.4 Problems with parallelism

Although parallelism is a very powerful way of speeding up rendering systems it suffers from some problems:

- Loss of coherence

  Many rendering systems gain speed from the use of *coherence*. These systems share processing work between pixels, primitives, objects or frames which are in some way related (for example by deriving the shading of one pixel from the pixel preceding it on the scan-line). Parallelism often separates primitives with similar properties by placing them on separate processing elements and thus it is more difficult to take advantage of the objects similar properties.

- Global state changes

  If a number of processors are processing a stream of primitives or vertices in parallel, the effect of global state changes (such as the current transformation matrix) can reduce the performance markedly. The way in which systems deal with such state changes depends on the way in which the global state is stored. One approach is to hold the state in a common section of memory which all processors can access. In this case a global state change requires that the system waits for all processing elements to finish computation on primitives which preceded the state change. The state memory is then updated and the system continues. This process is very costly due to the need to stall the pipeline. Another approach is to hold global state in each processor. In this case it is necessary to copy the state change information into the instruction stream to each processor. This increases the required bandwidth to the processors.

- Cost

  One benefit of parallelism is that a number of low cost, slower, processors may be used as an alternative to one expensive, fast processor. Although this may be beneficial, the cost advantages may be influenced by the cost of the support hardware required for each processor. While the sum of the cost of the cheap processors may be lower than the cost of the fast expensive processor much more support hardware is usually required in the parallel system.

A.7 Other approaches

This section describes a number of approaches to generating three dimensional images which are completely different from the approaches outlined earlier in this thesis. Most of the methods described in this section are presently not used in interactive systems due to the amount of computation required.

A.7.1 Recursive ray tracing

Recursive ray tracing is a method which performs global illumination calculations - that is to say that reflections, shadowing and refraction caused by objects in the scene, influence the shading of other objects; this allows the simulation of reflection, refraction and shadows. It is first necessary to consider the ray tracing process as the invocation of a subroutine which takes a description of a ray and returns an illumination value which corresponds to the illumination seen when looking in the direction of that ray.

This algorithm proceeds in the same manner as the ray casting method described above until it is time to evaluate the lighting expression for a pixel. At this time at least three more rays - one or more shadow rays, the reflected ray and the transmitted ray, are ‘shot’ from the point of intersection. These rays are known as secondary rays (the initial rays are the primary rays) and do not play any part in the hidden surface calculations. Recursive calls to the ray tracing procedure are made to calculate the illumination due to the reflected and transmitted rays. The illumination from these rays are then combined with the illumination found using the conventional lighting equation to form the illumination due to the primary ray. A limit is normally put on the depth of recursion - this ensures that the process will eventually finish.

The shadow ray is shot towards the light source (or in the case of multiple light sources one shadow ray is shot to each light source). If the ray intersects an object before reaching the light source then the intersection point of the primary ray is in shadow from that light and the effect of the light can be removed from the lighting calculations of the primary ray intersection point.

The reflected ray is shot from the intersection point at the same angle from the surface normal as the angle from the normal to the view vector. This simulates specular reflection. The transmitted ray is used to simulate refraction.

Most ray tracing systems support spheres as the basic primitive because the intersection test and calculation of normals is particularly easy. Other primitives, such as planes, and various general curved surface representations are also supported in many packages.

The simple ray tracing algorithm described above requires large amounts of computation. For a scene consisting of ‘n’ objects, and an image with ‘m’ pixels, ‘m x n’ ray intersections must be performed for just the primary rays - many more will be due to the secondary rays. For a model with just one thousand objects and a typical work station display with around one million pixels this requires well over a thousand million ray object intersections. For this reason ray tracing scenes may take many hours of calculation. Space subdivision methods (such as the octree, and grid cell subdivision described in section **5.5.2.1**) can be used to remove many of the intersection tests. **[KeHu94]** shows that the use of subdivision techniques can reduce the time taken from many hours to hundreds of seconds.

Since the calculations for each ray are essentially independent it is possible to execute the calculations for each ray (or a group of rays) on a separate processor in a multiprocessor computer. Careful management of communication overheads and processor utilisation can lead to near linear scaling of ray tracing algorithms on parallel computers **[KeHu94]**.

The recursive ray tracing method only handles some inter-object interactions. It handles specular reflection particularly well (which accounts for the reason that many ray traced scenes consist of shiny objects) but it does not handle diffuse reflection, shadows caused by reflected light or the correct shadowing of transparent objects.

When combined with texture mapping and motion blur recursive ray tracing can produce highly realistic images.

A.7.1.1 Hardware acceleration for ray tracing

Much of the research in ray tracing is being carried out on general purpose multiprocessor computers, however there have been a number of systems designed especially for ray tracing. These divide into hardware architectures for accelerating intersection calculations for particular primitives (e.g. **[Schn88]**) and multiprocessor architectures using general purpose processors which have been organised especially for ray tracing (e.g. the MARTI architecture **[HMSG92]** and the Pixel Machine **[PoHo89]**). In addition it is also possible to use conventional rendering hardware to accelerate the processing of the primary rays. To do this each object is assigned a unique identifier and the object is then rendered using a conventional Z buffer algorithm (for example) but its identifier instead of its colour is stored in the colour buffer. It is then possible to scan the colour buffer to find the first object which intersected the ray representing each pixel.

A.7.2 Radiosity

Radiosity is another algorithm which performs global illumination calculations. In some ways it has the opposite characteristics to ray tracing; it produces very realistic renderings of diffusely lit environments but does not cope well with shiny objects.

'Radiosity' is the rate at which energy (such as light or heat) leaves a surface. In the radiosity method, the scene is split into thousands of planar 'patches'. The aim of the algorithm is to work out a radiosity value for every patch - this can then be used as the intensity of the patch when rendering it. One method of doing this is to form and then solve, a series of simultaneous equations relating the radiosity of a patch to the radiosity of each other patch. These equations take into account the area of one patch visible from the other and the angle between the patches; this is represented by a series of *Form factors*. The calculation of the form factors and solving the equations can take many hours of computation. In addition the standard solution method uses a large amount of memory since it requires that all the Form factors are stored during the computation.

Having solved the equation (once for each colour component) the scene is rendered with a conventional rendering algorithm such as that described in **Chapter 3**. The patch radiosities are post processed to form vertex radiosities by averaging the radiosity of adjoining patches and the result used to provide the vertex colour for Gouraud shading of the patches. The radiosities calculated by this algorithm are completely view independent and thus the lighting stage of the rendering pipeline can be removed completely - therefore after a large amount of initial calculation the actual rendering stage is faster than conventional rendering.

The view independence of the radiosity approach is useful for architectural walk-through systems where the scene is completely static - however if any of the geometry in the scene is changed then it is necessary to recalculate all the form factors and solve the radiosity equation again. Furthermore in the conventional radiosity algorithm no image is available until all calculations have completed.

A progressive refinement approach to radiosity is presented in **[CCWG88]**. The progressive refinement approach reorganises the solution process so that a recognisable, but inaccurately lit, scene is produced quickly. Refinement of this model can then take place while the user is examining the present solution.

The progressive refinement approach is still of little use to VR systems since it is still difficult to modify the geometry of the environment without restarting the calculations. *Incremental radiosity* **[Chen90]** (which is a modification of the progressive refinement algorithm) allows modification of the geometry of the environment. The radiosity of the scene is then updated over a period of a number of frames. Although this does allow the use of radiosity in interactive design systems the updating algorithm produces some unusual (and potentially disturbing effects). In **[Chen90]** an office scene is shown in which a chair is moved. The lighting of the chair itself is updated almost immediately however the shadow of the chair stays still, the shadow is then seen to move over to the new position of the chair. This process took thirty seconds on the equipment used by Chen. Chen makes a comment which is important to all rendering systems which perform some preprocessing:

> *"The short turnaround time provided by this method is essential in an interactive image synthesis system, where the user should be encouraged to make changes instead of being penalised for it."* **[Chen90]**

Radiosity is still too slow to be used in a VR system where the user is allowed to change the geometry of the scene; it has, however, been used to calculate the lighting in architectural walk-through systems. Radiosity is also not a complete solution to the problem of lighting since its calculations are view independent and thus can not take account of specular reflection. Various extensions to radiosity are described in **[Fole90]** which account for specular effects.

A.7.3 Volume rendering

Some input devices (such as Magnetic Resonance and Computed Tomography medical scanners and the Confocal Microscope) produce a three dimensional array of values; each representing some property of the material being scanned such as density or transparency. There are typically many millions of cells (called *voxels*) in the array and thus one of the biggest problems with the technique is storage of the data. Octree encoding (described in section **5.5.2.1**) can be used to compress the data.

Early systems just displayed the data as a series of two dimensional ‘slices’ through the data set. It is difficult to observe structure from a series of two dimensional slices and so a number of three dimensional viewing methods are used. One of these methods produces a polygonal model based on an area of data with the same value (an *isosurface*). This can be used to produce a surface representing all the bone (or other substance) in a scan. These algorithms generate very large and complex polygonal models (often with hundreds of thousands of facets) which are normally only capable of showing one material type in the data set. It is possible to generate a number of such models for different objects in the scene (e.g. one for bone, another for blood vessels) and display a combined representation.

Another way of rendering volume data is to use ray casting. The simplest form of ray casting reduces to traversing the ray and finding the first cell whose value is above a threshold and is therefore taken to be opaque. Shading information can then be derived from the value stored at the cell the ray hit. More complex forms of ray casting continue the ray past the first partially opaque cell and accumulate the density values that the ray passes through. Novins et. al. show an efficient way of organising the ray casting process so that it accesses memory efficiently **[NoSG90]**. Hardware volume rendering systems such as the PARCUM II system **[JaSt88]**, and the 3D Viewing and Rendering Processor of the CUBE system **[KaBC91]** use specialised memory organisations to allow fast access to voxels along the rays.

In **[Kauf87]** Kaufmann suggests that voxel based systems could be used for the display of geometric objects. He outlines the CUBE system which includes a 3-D Geometry Processor which scan converts geometric objects into a voxel representation in the voxel buffer. In this way geometric shapes can be combined with voxel data gathered from medical scanners and similar devices.

A.7.4 View interpolation

View interpolation **[ChWi93]** operates on a large number of precalculated views of the scene. These images are stored together, with extra information which relates the different views. View interpolation takes a number of views and uses morphing to interpolate between them to approximate a range of views between the source images. One of the main advantages of this technique is that the time to generate each frame is independent of the scene complexity. In general, arbitrary modification of the scene geometry is not allowed. The designer of a scene can however specify a number of allowable changes - every additional possibility increases the amount of pre-calculation (and storage) required drastically.

In **[ChWi93]**, Chen and Williams describe various optimisations to reduce the storage requirements and time taken to generate images. In that paper, they show the results of a software implementation which generated frames (of 256 by 256 resolution) in under a fifth of a second on a powerful personal computer. In the same paper they describe the use of the algorithm to accelerate motion blur calculations (used to reduce temporal aliasing), and shadow generation. They suggest that it is a useful technique for simple virtual reality applications requiring the ability to walk around a precalculated environment. Due to its predictable (and fast) image generation times and its ability to represent complex scenes which have been rendered with realistic rendering methods, it would probably be suitable for VR applications if it could be extended to allow more flexible modification of the scene geometry, or if a means could be found to combine it with other techniques for the dynamic objects in the scene.

A.7.5 Combinations of techniques

A number of systems attempt to combine the benefits of conventional rendering, ray tracing and radiosity. **[Lera87]** describes a system which combines polygonal models rendered with a Z buffer and ray traced images. To do this the results of the ray tracing are plotted into a Z buffer using the Z position of the ray/object intersection. It is also possible to add extra objects to scenes whose lighting has been calculated with radiosity.

Many of these combinations are of limited use because the human visual system can easily detect which objects have been rendered with each technique. One example that the author has seen is a carefully shaded (and shadowed) room onto which moving objects were overlaid. This looked unconvincing since it was obvious that the few moving objects did not cast shadows correctly while other objects did. If the entire scene had been rendered without shadows the result may have actually been more convincing.

A.8 Errors and inaccuracies

This section describes a number of problems which reduce the quality of the images generated by rendering systems. Some of these problems are avoidable (often through the use of more complex and expensive processing) while others cannot be completely eliminated but their effect can be reduced. When designing a rendering system for a virtual reality application it is important to discover which errors are acceptable and which methods of reducing errors produce a high enough quality image for the application; for example when considering what type of spatial antialiasing is needed (or if any is needed at all) a simple supersampling method may be deemed sufficient even though it does not solve the problem completely.

A.8.1 Temporal distortion due to sequential scan-out

One consequence of the sequential scan-out process used on most display devices described in section **2.2**, is that different parts of the display are refreshed at different times. In a display device using the NTSC display standard, the display of one field of the image takes nearly 17 milliseconds, and the refresh of an entire frame takes more than 33 milliseconds. Thus the display of the last pixel of the last scan-line occurs more than 33 milliseconds after the first pixel of the first scan-line.

All the rendering techniques described so far render the entire image based on the state of the model at one moment in time. This is inconsistent with the scan-out process described above. In practice, this inconsistency appears as a distortion of the image in a scene which is moving relative the user. Consider the production of a sequence of images showing the viewer moving at speed past a lamp post. A conventional rendering system would produce a frame consisting of a vertical bar representing the position of the lamp post at one point in the frame (typically the start). Figure A.9 shows the effect of a viewer observing the post while moving to the right. The bar to the left of this image shows the position of the post relative to the viewer as the image is displayed. By the time the last scan-line is displayed the position of the post relative to the user is to the left of the displayed image. The viewer compensates for his movement and thus the straight bar now appears to be sloping in the opposite direction to the actual path of the post relative to the viewer **[MiBi93]**. Mine and Bishop in their paper **[MiBi93]** show that for typical speeds of movement, on an NTSC display, errors of around 30 pixels can be generated. This is especially problematic in the augmented reality systems that Mine and Bishop are concerned with.

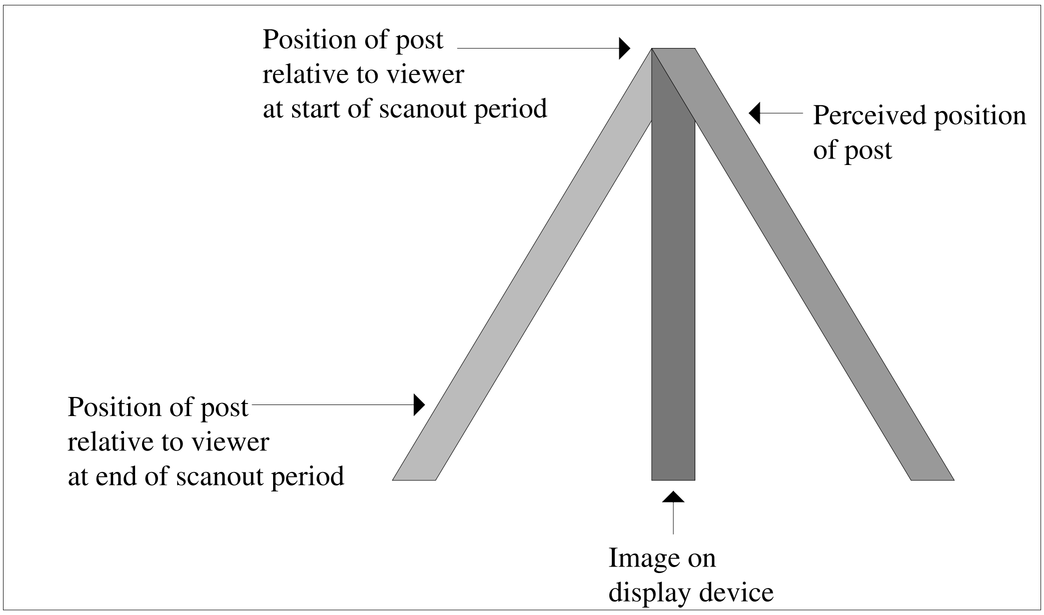

Figure A.9: Temporal distortion due to serial scan-out

To solve this problem fully it is necessary to render each pixel based on the viewing position and model description relevant at the time of its display. Mine and Bishop call this technique *Just-In-Time pixels*. This is impractical in reality due to the limited update rate of position sensors, the latency of the system and the vast increase in computation needed to compute each pixel based on a different model description and viewing transformation.

One approximation that can be used to reduce the amount of computation required, is to recalculate the model and viewing transformations on each scan-line rather than at each pixel. Although this still leaves some error Mine and Bishop show that, at NTSC resolutions and with realistic movement rates, the error is likely to be less than one pixel.

Mine and Bishop describe an implementation of Just-In-Time pixels using the Pixel-Planes 5 architecture; this is described in section **6.1.7.5**.

A.8.1.1 Spatial aliasing

Representing an arbitrarily complex image using a finite number of pixels leads to some loss of information. The rendering process incorporates at least two types of *sampling* which take continually varying signals and represent them using a discrete set of values. One of these sampling steps is temporal sampling, which represents a continuously varying image using a discrete set of frames; this process leads to *temporal aliasing* which is described in section **4.2**. The second sampling step is a spatial sampling which represents an image with a discrete number of pixels; this process leads to spatial aliasing. The causes of aliasing (and many ways of reducing the problems it causes) are described in detail in chapter 14 of **[Fole90]**.

Spatial aliasing leads to two separate types of visual anomaly. The first type is shown in Figure A.10. Due to the limited number of pixels available the sloping edge is represented by a rough ‘staircase’ like arrangement of pixels. Although images with this form

Figure A.10: Jagged edges caused by spatial aliasing

of aliasing are perceived to have a low image quality, Booth et. al. show that in static scenes using high resolution displays, the aliasing does not reduce the performance of participants in simple observation experiments **[BBCM87]**. In virtual reality systems where the position and orientation of images changes frequently this type of aliasing can be particularly problematic since the steps on the edges move from one frame to another. This movement is extremely noticeable and disturbing and it can also provide anomalous cues to the viewer. One such cue could be that a participant in a driving training system could learn to tell the speed of another object from the rate of movement of the steps on the edges of the object; although this may help the participant in the training system they will find it more difficult when faced with driving in the real world.

The second type of anomaly due to spatial aliasing is shown in Figure A.11. The rendering algorithms previously described fill a pixel when a certain point in that pixel (typically the centre) is covered by the primitive. Very thin primitives (such as the one shown in the figure) may fit between the pixel centres and therefore not get rendered at all. This effect is exaggerated in VR systems since the primitive will sometimes move across pixel centres and appear and at other times disappear. One of the most common examples of this problem can be observed in simple flight simulator systems, where the wings on an aircraft flying into the distance flash on and off as they move relative to the pixel centres.

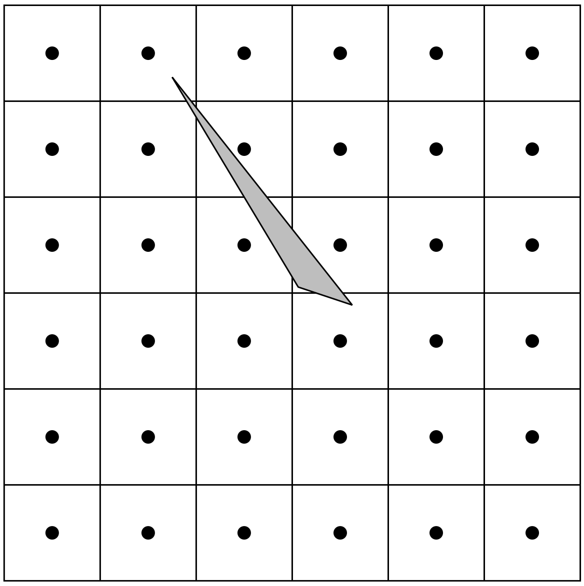

Figure A.11: Loss of primitives due to spatial aliasing

Both of these problems are alleviated at higher resolutions, but may still be noticeable on the highest resolution displays commonly in use. Various methods are used to reduce spatial aliasing problems, some of which will be described here.

A.8.1.2 Supersampling

Supersampling is one simple (but often expensive) method of reducing spatial (and temporal) aliasing. The conventional rendering method can be thought of as sampling the scene once per pixel at the pixel centre. When supersampling is used each pixel is sampled a number of times and the results are later combined to produce the final pixel colour. A number of different sampling patterns are in common use; the simplest pattern is just a grid (of perhaps sixteen samples arranged in a four by four pattern) which corresponds to multiplying the resolution along each axis by a set amount. This is expensive since for a subpixel grid containing sixteen samples per pixel, sixteen times as much rendering is required, even on pixels which do not contain an edge. Another method is *adaptive supersampling* in which extra samples are generated on pixels around areas likely to suffer from aliasing. Combining the samples may be done using a simple mean or it may be weighted such that samples near to the pixel centre are given a larger influence over the final pixel colour. Supersampling can not completely eliminate these aliasing problems, however the increased resolution can help reduce their seriousness.

A.8.1.3 Coverage masks

Another method of antialiasing is to use the proportion of the pixel covered by a primitive to determine how much that primitive contributes to the final pixel value. Determining the coverage value can be done analytically from the geometry of the primitive, however this method is expensive and is not easy to integrate into conventional span filling systems[1]. An alternative method of producing coverage information is through the use of *coverage masks*. As in the supersampling method the geometry of each primitive is examined at more than one point within each pixel; however the colour and (possibly the depth) information is only calculated for the centre point. Figure A.12 shows an enlarged view of one pixel at the tip of a triangle and sixteen sampling points within that

1. Analytical calculation of coverage information is often easier to implement in systems which use scan line methods for hidden surface removal, such as those described in section **A.2.2**.

pixel. On the left of the diagram is the coverage mask; a one is present if and only if the primitive covers the corresponding sample point. In the simplest implementations the coverage mask is used to index into a precalculated table which contains a coverage value for each mask pattern. This coverage value is then used to blend the contribution from this primitive with the present pixel colour using alpha blending.

The conventional means of generating coverage masks still allows primitives to disap-

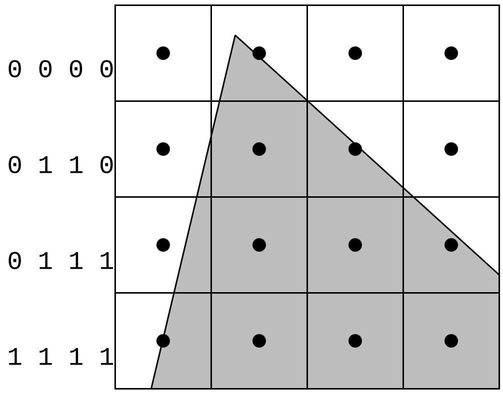

Figure A.12: A coverage mask

pear between the subpixel sample points; for example a thin primitive (seen almost edge on) moving vertically downwards will often contribute nothing as it lies between the rows of subpixel sampling points and will suddenly contribute four samples as it moves over the row of sampling points. A more accurate method of generating the coverage masks is described in **[Schi91]**. This method generates a coverage mask which has a proportion of bits set which is similar to the proportion of the pixel covered by the primitive. The algorithm uses information from the scan conversion section of the pipeline to discover how close edges are to pixel centres and the direction of slope of edges to discover which bits in the subpixel mask to set. Many different antialiasing techniques exist; further detail can be found in **[Fole90]** and **[Akel93]**.

A.8.2 Z buffer errors

One problem with the Z buffer method of hidden surface removal is that the Z buffer has a limited resolution, and thus two surfaces which are extremely close together may share the same Z value. When this occurs the wrong surface (or more often part of the wrong surface) may be displayed. This is very disturbing (and wrong) and is complicated by the fact that in a borderline case the surface which is visible at each pixel changes depending on the slightest change in viewing parameters, causing whole surfaces to flicker from one frame to the next.

Z buffer errors occur most often when two surfaces touch each other; for example in one scene containing a waste paper bin resting on a floor, the floor was visible through a large section of the bin's base. These problems are inherent in Z buffering since the base of the two objects effectively have identical Z coordinates. This problem is similar (but not identical) to the problem which occurs when two objects are close enough to use the same Z value due to the limited accuracy of the Z buffer. One final problem which Z buffer quantisation can lead to, is the ragged joining of primitives at an edge.

A solution to the problem of limited accuracy, is to place a small gap between objects which would otherwise touch in the model. This can often resolve the problem but at the risk of allowing a gap to be visible under certain viewing conditions[1]. Another solution is to include priority flags in addition to the Z buffer, such that high priority items always overwrite lower priority objects which use similar Z values. This technique could be used to ensure that polygons representing the dashed line down the middle of a road are always visible and not obscured by the road surface.

Z buffers in rendering systems are often 24 or 32 bits deep and thus can represent between 16 million and 4 000 million discrete Z values. Since these buffers have such a large dynamic range it is often difficult to understand why Z buffer aliasing problems occur so frequently in objects which do not touch in the model description. If the entire range of a 24 bit Z buffer was used to represent the position of objects in a room ten

1. The author has seen one system where a door handle could be seen to float a small distance in front of the door if observed closely.

meters deep each Z buffer value would represent less than one micrometer and it would therefore appear that Z buffer quantisation problems should not occur. However, as mentioned in **Chapter 3**, the raw Z value is not actually stored in the Z buffer and instead the reciprocal of the Z value is stored. This reciprocal measure distributes the resolution of the Z buffer unevenly in space such that a unit change in the stored value represents only a small change in distance in an object close to the viewer but a much larger change in an object far from the viewer. For this reason the high dynamic range of the Z buffer is not equally effective over the whole scene.

A.8.3 Subpixel accuracy

A number of different anomalies are attributable to the rounding of vertex coordinates to the integer pixel coordinates. In simple systems, the vertex coordinates are rounded to integer values immediately before the rasterisation stage; since each vertex of a primitive may be rounded by a different amount the model is distorted. This rounding also changes from frame to frame as the viewpoint is changed causing the distortion to change rapidly; this effect has been observed by the author in a scene which contained a chequered floor which was distorted differently in each frame, making it appear as if the floor was moving. Another similar effect occurs if the vertices of two touching primitives are rounded in opposite directions; usually due to slight differences in their position caused by mathematical inaccuracies. In this situation, two vertices which should be coincident are rounded to different pixels leaving a gap (or in some cases an overlap) between the two primitives.

Lathrop, Kirk and Voorhies demonstrate one method of avoiding these problems and highlight a related problem which is often mistaken for being caused by quantisation in the Z buffer **[LaKV90]**. When vertex coordinates are rounded, the parameters defining depth and colour information which were defined at the original unrounded position are often applied to the rounded position. The change in depth can cause primitives which should be hidden to become visible through other primitives or the areas where primitives join to appear ragged. Correcting for this inaccuracy is achieved by using linear interpolation to calculate the depth and colour information at the rounded vertex position; this process is expensive to implement. The method presented in **[LaKV90]** provides a solution to this problem using only integer arithmetic.