Chapter 3

Conventional rendering techniques

This chapter describes a simple conventional hardware rendering system from the host to the final image on the display device. It describes very few of the algorithms available to the designer of a rendering system; instead it provides a reference to which other systems described later can be compared. During the descriptions, the methods used in various existing systems are discussed. Full descriptions of many of the algorithms are beyond the scope of this thesis, further details can be found in **[Fole90]**.

3.1 The Rendering Pipeline

The conventional way of representing the rendering process is as a pipeline; one version of this is shown in Figure 3.1. This is based on the '*Rendering pipeline for z-buffer and Gouraud shading*' on page 807 of **[Fole90]**, with the rasterisation section and display section shown in more detail, and the frame buffer and tessellation section added. This is one possible arrangement of the pipeline - tessellation, lighting and various other sections can be moved depending on the particular rendering scheme employed. The rasterisation engine and frame buffer arrangement, in particular, are highly specific to the scheme described in this chapter.

3.2 Model traversal

In most systems there will exist a description of the entire environment - known as the model. There exist two main techniques for processing this; the first is known as *retained mode*, in which the graphics system holds a copy of the model in memory local

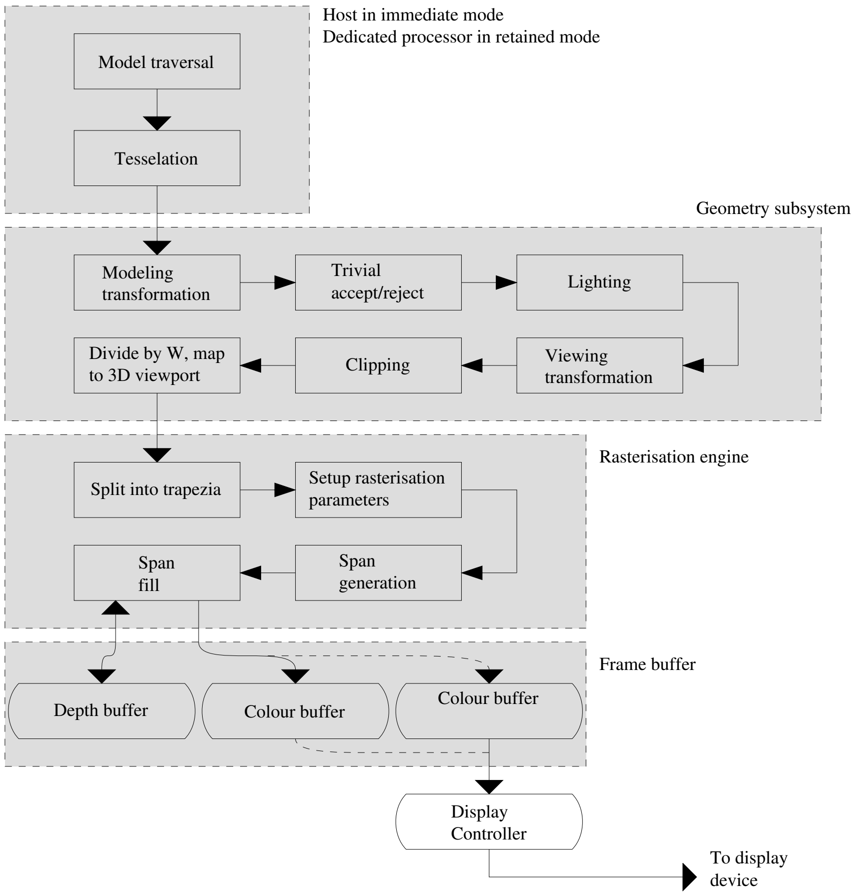

Figure 3.1: The rendering pipeline

to the graphics system. It is then the graphics system's responsibility to traverse the model and extract primitives to be processed by the rest of the system. The host communicates with the graphics system by issuing commands which edit the current database.

The other technique is *immediate mode*, in which the host must supply a list of primitives to the graphics system for every frame. Immediate mode is considerably more flexible since software running on the host can emulate a retained mode system while the converse is not possible. In an immediate mode system the model of the environment could actually be hard coded into the structure of a program - for example a subroutine which drew a table or other object would actually generate the primitives at the time of rendering.

Many systems restrict the primitives in the model to being polygons since these are particularly easy to rasterise and shade. Often only triangles are supported since they are guaranteed to be planar and have a small fixed number of vertices which makes much of the later hardware a lot simpler[1]. Storing pure polygonal models consumes large amounts of memory and is an unnatural description of an object such as a sphere or cylinder. To remove this problem many systems store higher level objects such as curved surface descriptions in the database. It is then necessary to *tessellate* the surface into polygonal areas. The process of tessellation may be performed by software on the host, or by software or dedicated hardware in the rendering system.

At this stage it is useful to compare *immediate* and *retained* modes. Immediate mode requires a large amount of processing by the host; which may already be heavily burdened by work necessary to maintain the model (such as collision detection, dynamics simulation, processing of input devices and managing tactile feedback devices). The primitives generated by the host must then be transferred to the immediate mode rendering system - which requires a very high bandwidth link. Consider a scene completely described using separate triangles. Each triangle requires that three vertices are transmitted to the renderer, each consisting of an x, y and z position (4 bytes each), and the three colour components (1 byte each). This gives a total of fifteen bytes per vertex and thus 45 bytes per triangle. For a scene consisting of ten thousand triangles (not an unusual figure), this requires 450 000 bytes per frame, equivalent to over 12 MByte per second (at 30 frames per second). This figure is too high for common peripheral buses such as SCSI (with a bandwidth limit of about 10 MByte/second for single width SCSI-2).

Retained mode typically requires little processing by the host and a lower bandwidth link; however if large sections of the scene change from frame to frame the bandwidth required may be *higher* than a similar immediate mode implementation because the

1. A collection of more than three points is not necessarily suitable for representing the vertices of a polygon since the vertices of a polygon must all lie in the same plane.

commands needed to edit the database are more complex than a simple primitive description. A disadvantage of retained mode is that the complexity of the scene is now limited by the amount of memory available on the rendering system. Often in retained mode systems, the host has to maintain a partial copy of the model for operations such as collision detection; the fact that a copy of the model now exists in two places may be seen as wasteful. However an advantage of retained mode over immediate mode, is that the rendering system has a lot more control over the way in which the scene is rendered.

3.3 Hierarchical models

Although it would be possible, it is not normal for the model to be represented as an unstructured set of primitives. Many systems represent the model as an acyclic graph. Figure 3.2 shows a model hierarchy for a simple model consisting of two rooms, each with four walls (each one being a polygon) and a light. The first room contains a table and a teapot, he second room is empty. The table in the first room consists of a top (a polygon), and four legs (each a cylinder). The teapot consists of a spout, handle and body each consisting of a number of surface patches. The legs of the table only have one definition but are used four times in the definition of the table. This saves memory, allows properties of all the legs to be modified at once, and may save time by sharing processing.

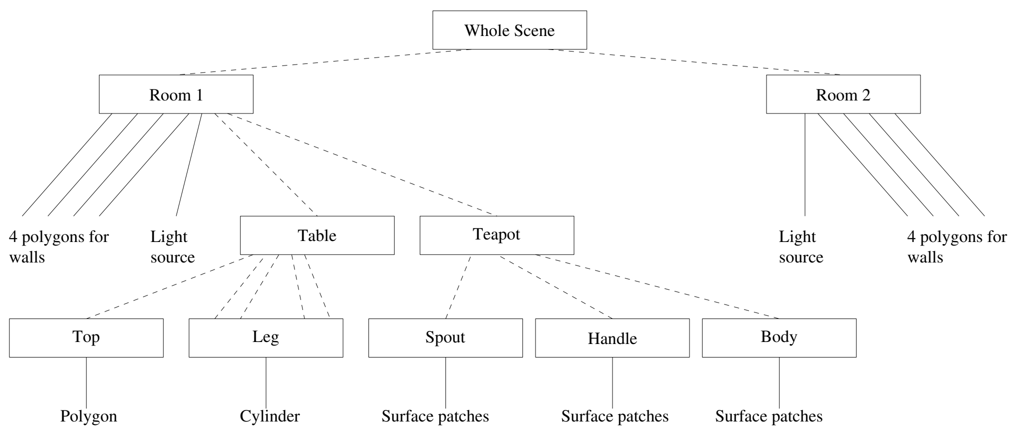

Figure 3.2: A hierarchical model

Within a hierarchical model each object has its own coordinate space. In the teapot, for example, the position of the spout, the handle and the body are specified relative to some position in the teapot (such as its centre). The surface patches making up the spout are then defined in the spout's own coordinate space. To make hierarchical models practical whenever one object is used in another, a transformation is specified that positions the child object inside the parent's coordinate system. In the scheme which Figure 3.2 depicts, a transform exists on every inter-object link (shown dashed). Each primitive (such as the cylinders, polygons and surface patches) contain their own position information. The transforms consist of rotations, translations and scalings to position and scale the object in its parent's coordinate system. An alternative approach is to define objects such as cylinders to be a particular size and orientation (typically one unit high and long) and then use transforms to position those primitives in the world.

Each primitive contains a set of attributes including its surface properties, positions of vertices which define the object, and normal vectors (which will be used later for lighting and culling).

The exact form of the model structure and the choice of retained or immediate model is often determined by the graphics library in use. Libraries such as PHIGS use a hierarchical description and are suited to retained mode systems. Other libraries such as OpenGL are much more suited to immediate mode systems. The pros and cons of various graphics libraries are discussed in more detail in section **6.2.2**.

3.4 Tessellation

For objects such as cylinders, tessellation can be done very simply (Figure 3.3); similar (but more complex) methods are available for spheres, cones and other geometric shapes. Curved surfaces may be defined parametrically using a variety of methods. Very few rasterisation systems attempt to display these directly and instead opt to divide them into polygons. The tessellation of parametric surfaces is commonly done by a recursive method where the surface is split down into a number of subsurfaces. This process continues until the surfaces are almost flat at which point the surfaces are replaced by polygons. For a detailed description of many types of surface patches see **[Fole90]** chapter 11.

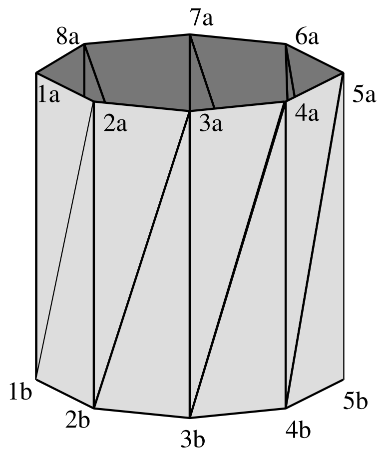

Figure 3.3: A cylinder represented by a series of triangles

Figure 3.4 shows three different representations of a sphere using increasing numbers of polygons. The more polygons that are used, the more like a sphere it appears - the shading on the cylinder also looks a lot more realistic with more polygons (although as will be seen below, techniques exist which allow realistic shading effects with a smaller number of polygons). At this stage in the pipeline we have no knowledge how large the object will appear on the display, it is therefore difficult to know how many polygons will be needed before the representation looks sufficiently realistic. This problem is one reason that tessellation is sometimes placed further along the pipeline.

3.5 The modelling transformation

The transforms described in section **3.3** on hierarchical models constitute the *modelling transform*. Each object has its own coordinate space within which its children are defined - these coordinate spaces are known as *modelling coordinate system*. The whole

60 polygons 144 polygons 264 polygons

Figure 3.4: Three tessellations of a sphere

scene has its own coordinate system known as the *world coordinate system*. The transforms used to move between these coordinate systems consist of rotations, translations and scalings. These transformations are all *affine transforms* - that is lines parallel in an object prior to transformation are still parallel afterwards. Affine transformations can easily be represented in a 4 x 4 matrix form using the *homogeneous coordinate system* **[Fole90]**. Although this adds an extra value to each vector, it allows translations, and perspective transforms to be included in the matrix. When represented in matrix form the act of transforming a point from one coordinate space to another consists of multiplying the transformation matrix by the vector representing the point. The main advantage of the matrix representation is that a number of transformations can be combined into one matrix simply by multiplying the transformation matrices together. This represents a considerable saving when compared to applying each transformation separately.

As the hierarchy is traversed the *ModelToWorldMatrix* is updated to reflect the transformations for the present subtree. A stack of matrices is needed to allow the previous matrix to be recovered as the algorithm returns back up the tree. Unfortunately the normal vectors associated with the primitives can not be transformed using the same matrix; this requires a separate matrix to be maintained during traversal.

A full description of matrix methods for graphics can be found in **[Fole90]** chapter 5.

The output of the modelling transformation stage is a stream of triangles with each of the vertices defined in world space. Together with each face or vertex there exists a normal vector used for lighting effects and back face culling. On an object such as a cube which is naturally polygonal, the vertex normals correspond exactly to the surface normals, however on a curved surface, the vertex normals are usually the result of averaging the surface normals of the faces meeting at that vertex. In some systems this averaging is done during traversal, in others it is performed during model creation.

If the vertices of a face are stored in an anticlockwise order (or at least the order is consistent) it is possible to derive the surface normals from the vertex coordinates and thus avoid the need to store and transform them at this time.

3.6 Trivial accept/reject

One of the best ways of speeding up a rendering system is to ensure that any objects that the user can not possibly see are not rendered. In general methods which attempt to remove non-visible polygons from a scene are called *culling*.

A simple technique called ‘back-face culling’ can be implemented at this stage. If an object is a polyhedron (for example a cube) approximately half the faces will be facing away from the user and will be hidden; back-face culling removes these hidden faces. This technique is not suitable for an object which allows any part of its interior to be viewed - an example being a cube with the top face missing (e.g. an open box).

The normals to the surface of the polyhedron could point in to or out of the polyhedron. By standardising on the normals being defined so that they point out from the polyhedron, the front and back of each face can easily be identified (Figure 3.5). From this figure it is easy to see that the two back faces and the base of the cube (whose normals point away from the viewer) are hidden by the other faces.

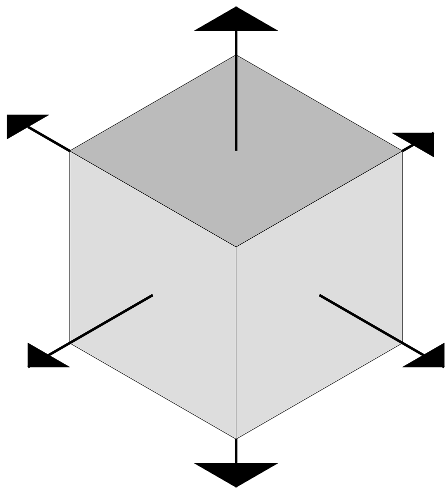

Figure 3.5: Cube with surface normals

This calculation is computationally intensive and benefits from the fact that in most scenes approximately half the faces will be removed, so that the cost is greatly out weighed by the benefits in future stages. An alternative is to place the back-face cull test after the viewing transform. At this point the test is just a matter of testing the sign of the Z coordinate in the normal.

3.7 Definition of the viewing frustum

Another simple culling method which can be employed here is a test based on the viewing frustum. The viewing frustum is a volume in space which can be seen by the viewer - any object outside this space is definitely not visible. Figure 3.6 shows a basic model of the view frustum. The display device corresponds to the *view window* - which is a section of the *view plane*. The *Far clipping plane* defines the furthest object the viewer can see, and the *Near clipping plane* defines the closest object the viewer can see.

It is useful to remove all objects outside the viewing frustum before the computationally expensive lighting stage. If the model is much larger than a typical viewing volume (as is usually the case) then this culling reduces the work of later sections drastically.

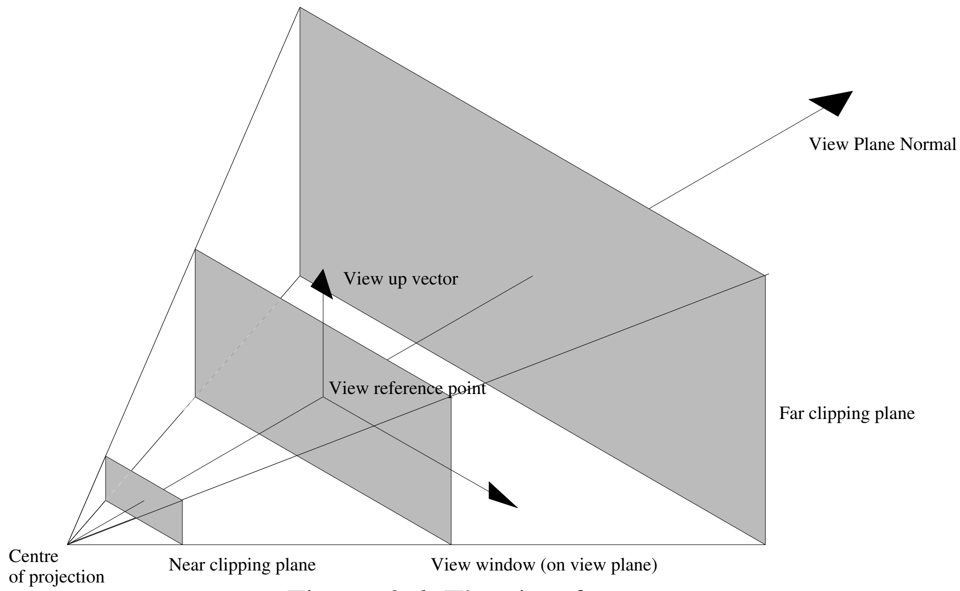

Figure 3.6: The view frustum

Unfortunately, testing whether a point is in a frustum is a reasonably intensive task. One solution is to cull to a more conveniently shaped volume (such as an axis aligned cuboid) which is just large enough to fit the viewing frustum into. This shows an important point of the trivial rejection stage - it does not have to reject all objects which can not be seen. As long as no potentially visible objects are lost, the accuracy of the trivial rejection stage may be determined by the designer of the system. These issues are described in more detail in sections **5.5** and **7.1**.

3.8 Lighting

The lighting stage assigns colours to faces or vertices based on the properties of the face involved and the properties of any light sources in the scene. The scheme described here is one type of *local illumination model*; where the colour of a face is determined solely by its own properties and the properties of the light source. This technique is used in almost all real time rendering systems because it is considerably simpler and faster than *global illumination models* which take into account interactions with other objects.

The model that will be described is the Phong lighting model (as described in chapter 16 of **[Fole90]**). This lighting model is not an accurate model of the physical behaviour of light - however it works well and is fairly cheap to compute.

The model builds up the illumination out of three components:

- Ambient illumination

  The ambient illumination is constant irrespective of viewer position, surface normal or light source position and is dependent only on surface properties and the amount of ambient light in the environment.

- Diffuse illumination

  Diffuse illumination produces a dull, matt lighting effect. The intensity of diffuse illumination is dependent on the angle between the light source and the surface normal, as well as the surface and light properties.

- Specular illumination

  The specular illumination of an object accounts for bright highlights on the surface of an object and is used to give a surface a shiny appearance. The specular contribution is based on the angle between the viewer and a reflection vector representing the direction in which the light reflects off the object.

This model can be used for scenes with multiple light sources, where the illumination is the sum of the ambient term and a diffuse and specular term for each light source. The entire lighting model must be applied for each colour component of the colour system in use.

Since the specular and often the diffuse terms vary across a surface, it is desirable to evaluate the lighting model at each pixel; this is the approach taken in the Phong shading method[1] (described in section **5.4.3**). Evaluating the model at each pixel is however costly. The solution described below (Gouraud shading) provides a much faster, less realistic compromise, by interpolating the results of the lighting model calculated at each vertex, across the surface.

3.9 Viewing transformations

This stage consists entirely of generating and applying one transformation matrix which incorporates:

- A translation so that in the resulting coordinate space the eye is at the origin.
- Rotation accounting for view orientation
- Scaling the view volume for convenient clipping

  Clipping against a frustum is difficult. One method of simplification is to transform the view volume into a standard form, called the *canonical view volume*, in which the vertices defining the cube have more convenient values. For details refer to [**Fole90**] chapter 6 which details transformations for culling and the rest of the viewing transformation process.

- Perspective projection.

  The perspective projection makes objects which are further away appear smaller. Without perspective it is not possible to distinguish between two identical objects at different distances from the viewer. Figure 3.7 illustrates the principle. The point to be projected, P, forms one vertex of the triangle OPb which is a similar triangle to OP'a.

  Using the similar triangle technique, x' (the x component of the projected point P') can be calculated as:

1. Not to be confused with the Phong lighting model.

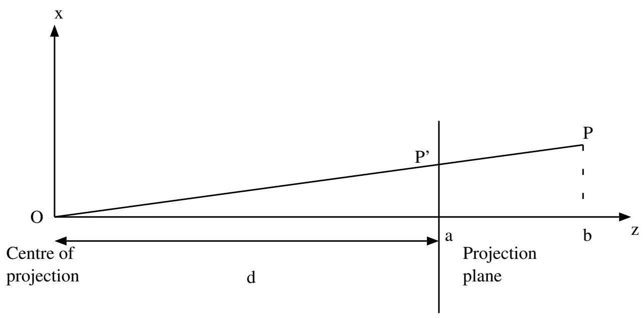

Figure 3.7: Perspective projection

$$x' = \frac{dx}{z} \qquad \text{EQ 3.1}$$

Perspective introduces two divisions at each vertex; this is unfortunate since division is particularly expensive (and slow) to implement in hardware. The actual division is carried out later in the pipeline, after clipping.

Although this process is very complex, all the transformations may be combined into a single matrix and applied simultaneously. At the same time the clipping will have been simplified.

3.10 Clipping

Using the viewing transformations above it is now simple to test whether a point is outside the viewing volume. Clipping of an entire polygon is often performed using the Sutherland-Hodgman polygon clipping algorithm **[Fole90]**. This algorithm works on a stream of vertices, adding, modifying or removing vertices as the clipping is performed. The algorithm is applied separately for each plane clipped against.

The result of clipping is a polygon which must be converted back to a triangle to be rasterised using the simple algorithm below. Most primitives in a typical scene do not require clipping - this is fortunate since clipping is quite an expensive process.

In general rendering systems the application is allowed to specify extra clipping planes. In most VR applications it is only necessary to clip against the view volume. This can simplify the clipping since it is possible to optimise out the clipping against the right and bottom edges of the display. If part of the primitive is visible then to clip against the bottom or right edges it is possible just to stop rasterising the present trapezium or span when the edge is reached.

If clipping is performed after perspective division then it is possible for objects behind the viewpoint (with a negative z value) to be mapped back in front of the viewpoint. This occurs because the perspective division maps *all* points onto the projection plane. Although objects which are completely behind the view point can be removed in the trivial reject phase, large objects which are both in front of and behind the viewpoint (such as a wall which the user is standing next to) must be clipped correctly.

3.11 Perspective division and mapping to the viewport

At this stage in the pipeline, all the coordinates are now still in homogeneous coordinates, but now all fit into the viewing volume. This stage carries out the perspective division and scales the resulting output to the viewport.

It is possible to optimise the division process by noticing that the two divisions have the same divisor; it is thus possible to replace the two divisions by one reciprocal calculation and two, less expensive, multiplications.

3.12 Implementation of the geometry subsystem

The actual transformations and matrix multiplication require large amounts of computation - usually using either large fixed point or floating point values. Conventional microprocessors are capable of doing the required operations but the performance can be enhanced significantly by using specialised hardware, processors designed specifically for three dimensional graphics operations or digital signal processors which share many of the required characteristics.

The other operations that the geometry subsystem must perform (trivial accept/reject, lighting, viewing transformation, clipping and division by W and the mapping to the viewport) are also computationally expensive. The design of the geometry subsystem must include:

- Fast multiplication and fast multiply accumulate

  Matrix by vector calculation, matrix by matrix calculation and the dot product operations, all require large numbers of multiplications. A vector, matrix product requires a total of four multiplications and 12 multiply accumulates (or 4 multiplications and 9 multiply accumulates when no perspective transformation is involved). General purpose microprocessors typically take many cycles to complete a floating point multiplication, and a significant number of cycles to perform a fixed point multiplication.

- Rapid access to large amounts of data

  As well as performing a large amount of arithmetic for matrix operations a large amount of data has to be accessed from memory (16 matrix values, 4 values from the incoming vector and 4 values for the result). Typical microprocessors have between 16 and 32 registers. A processor with 32 or more registers which allows most of the registers to be used for computation is unusual, but when available could possibly handle the matrix by matrix case reasonably efficiently.

- Fast access to look-up tables

  A reciprocal look-up tables can be used as a substitute for division, similarly tables can hold values for sine and cosine. Fast access to these tables is useful.

- Communication with the rest of the pipeline

  Passing data to the next stage and receiving data from the previous stage can be a difficult job for a conventional processor. Building buffering control and communications links into the processor can help a lot.

Many early systems used custom hardware to handle all the geometry - in particular the 'Geometry engine' **[Clar82]**. The Geometry Engine was a simple arithmetic engine with four identical sections capable of arithmetic - ideally suited to handling the four components of a vector operation (or handling two for a more complex operation). A number of geometry engines were used together to form a pipeline handling all geometric operations in the graphics system. In more modern systems it is more common for off the shelf processors designed for digital signal processing or the Intel i860 processor (described in section **6.3.1**) to be used. Digital Signal Processors (DSPs) such as the Texas Instruments TMS320C30 **[Texa92]** provide fast multiplication, on chip fast access RAM, data transfer in parallel with computation, and fast loop instructions. This combination solves many of the problems listed above. DSPs tend to be available at low cost, fixed point versions are especially cheap.

Some manufacturers still prefer to create their own processors for the geometry system; the *Leo* system by Sun is one such system, which is described in more detail in section **6.1.2**.

3.13 The frame buffer and display controller

As discussed in section **2.2**, the display device needs to be refreshed constantly by the display controller. In conventional rendering systems the display is split up into an array of small squares (pixels) each of constant colour. The colour of each pixel is stored in a rectangular array called the *frame buffer*. Every time that the display is refreshed each pixel in the frame buffer is read out and passed to a digital to analogue converter to drive the display device. The frame buffer must support a high bandwidth connection to the display controller and a high bandwidth stream from the rasterisation engine; concurrently supporting two high bandwidth streams is difficult. Frame buffers are now generally built from *Video RAM (VRAM)*. VRAM has two ports - one which can be accessed in a similar way to conventional DRAM and another which is accessed in a serial fashion - normally just for read out. The renderer accesses the frame buffer through the conventional port while the display controller uses the serial connection. This removes the problem of shared access.

A problem which still remains is that if the same section of memory is being updated by the renderer as the image is being displayed, partially completed images will be displayed. The method of avoiding this problem in common use is to have two frame buffers - one which gets updated by the renderer while the other gets displayed. This method is called *double buffering*. Double buffering is conceptually similar to implementing a two entry pipeline of rendered frames, with one entry containing the new frame and the other the displayed frame. This system does incur extra latency of one extra frame time.

In Figure 3.1, the frame buffer is split into two colour buffers (which hold the double buffered images) and a depth buffer. The depth buffer implements a hidden surface removal scheme called *z buffering* - it is described in detail in section **3.14.3**.

3.14 Rasterisation

It is the job of the rasterisation engine to convert the primitives into a series of pixels in the frame buffer. This section will restrict itself to the rasterisation of triangles since they are the simplest primitives to handle. The rasterisation step handles a number of different tasks:

- Scan conversion

  Scan conversion is the actual process of discovering which pixels on the display are covered by the primitive.

- Shading

  Shading determines the colour of each of the pixels which have been selected in the scan conversion stage.

- Hidden surface removal (HSR)

  Hidden surface removal ensures that if a number of primitives cover the same pixel, the assigned colour is the colour of the primitive nearest the viewer. There are many HSR algorithms, some of which operate higher up the pipeline.

The useful property of the rendering method described below, is that these three tasks are handled simultaneously. The input to this stage from the geometry pipeline is a series of triangles. Each vertex has assigned to it a colour and an x, y and z coordinate; it will be presumed here that the x and y coordinates are measured in whole pixels; errors caused by this simplistic approach are described in section **A.8.3**.

The task is actually partitioned into two sections. First some preprocessing on the whole triangle must be carried out, then a large number of similar operations must be done to perform the actual rasterisation. In general the preprocessing is done in a microprocessor (often of similar design to the processor used in the geometry subsystem; in some systems the same processor performs both the geometrical operations and preprocessing for triangle rasterisation).

3.14.1 Scan conversion

The way many systems scan convert a triangle is to partition the triangle into two trapeziums - both of which have two sides which are parallel to the X axis (or are of zero length). This simplifies the requirements of the span generation hardware. This is just an application of a common computational requirement - simplify the inner loop at the cost of a one off complication. Figure 3.8 demonstrates the process of subdividing the triangle.

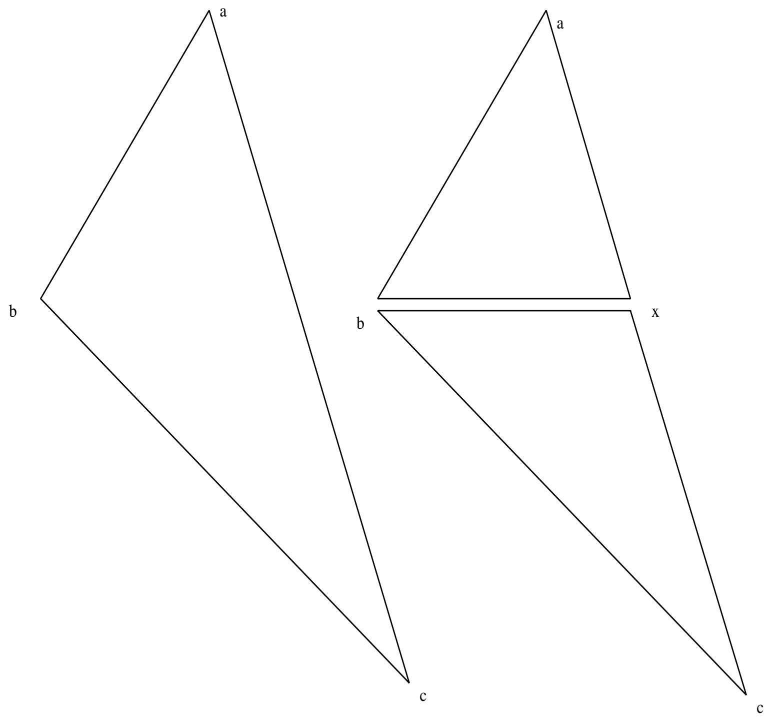

Figure 3.8: Splitting a triangle into trapezia

Span generation

A span is an *axis aligned run of pixels* - in most systems it is horizontal. The span generator produces a span for each scan line that the trapezium intersects, consisting of a start and stop X value corresponding to the intersection of the primitives edges with the scanline. One way to generate the spans is to solve the equation of the line representing the edge for the current Y position. This would be very expensive and difficult to implement in hardware. An easier method, which is simple to implement, is to calculate in the setup processor the slope of the edges and then add the slope value to the edge's current X value at each scan line (Figure 3.9).

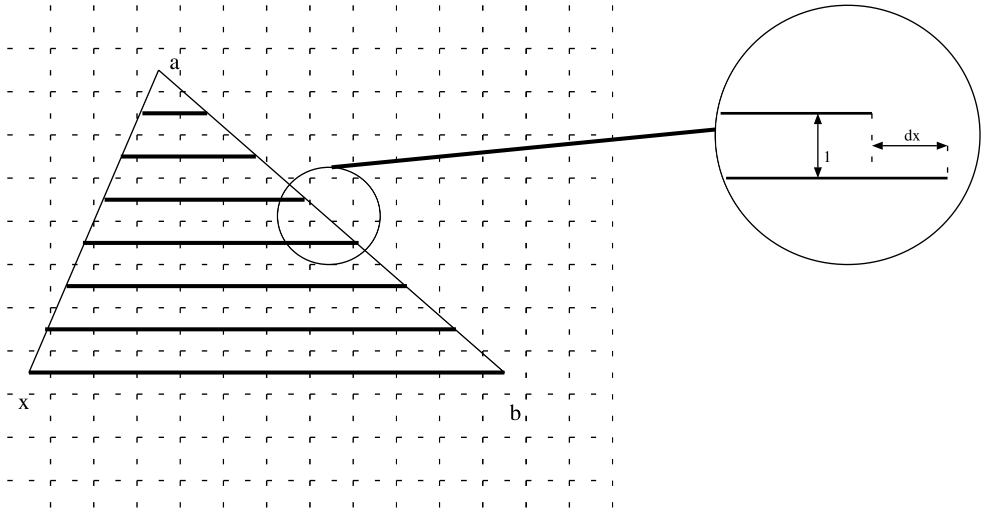

Figure 3.9: Splitting a trapezium into spans

Span generation is a simple iterative process; the easiest hardware implementations consist of an adder and a register for each edge together with control logic and a counter giving the height of the trapezium. On each span the slope is added to the current value and output to the span fillers. At the start of each trapezium the setup processor has to load initial x and y values, slope and the number of spans into the span generator. The amount of data that must be transferred between the setup processor and span generator is quite high; one method of reducing the bandwidth is to note that when rendering triangles which have been split into trapeziums, only some of the values need reloading at the end of the first trapezium.

Span filling

The span filler takes the spans from the span generator and writes a pixel at every location in the display memory from the start X position to the stop X position. One complication is that the spans are rarely multiples of whole pixels in length and rarely are their start positions aligned to a pixel edge. Simple truncation of edge coordinates produces gaps or overlaps between triangles forming the same surface. Rounding the start position up and the end position down with an extra condition that, if the end position has no fractional part it is decremented by one pixel, helps to solve the problem. Table 4.2 lists the start and end points of a number of spans of triangles which should but up against each other. An important point from the example is that no pixels are addressed by two consecutive spans and that no gaps have been left between the spans.

	raw start	raw end	modified start	modified end
a	2.3	5.8	3	5
b	5.8	7.2	6	7
c	7.2	11.0	8	10
d	11.0	14.5	11	14

Table 3.1: Examples of rounding in span start and end positions

In its simplest form the span filler just consists of a counter, which steps through video memory addresses corresponding to the pixels in a span, and logic to generate the control signals to the frame buffer and to stop it at the end of the span.

3.14.2 Shading

In the previous section the source of the colour that was written into the video memory was conveniently omitted. This colour value must be generated by a shading algorithm for every pixel in the triangle. The simplest method is *flat shading*, (demonstrated in Figure 3.10), where the same colour is used on the whole triangle. This method is very simple and cheap to implement, however the individual facets used to approximate curved surfaces are easily visible. In general for still images this does not look good, but the shape is still visible; however when objects are moving relative to the viewer the effects are much worse. The edges between the faces become much more noticeable and can dominate the image. An effect which was observed by the author when creating Figure 3.10 was that at certain rates of rotation, temporal aliasing can occur and the edges can appear to move in the opposite direction to the object; this effect is very distracting.

Figure 3.10: A flat shaded teapot

One of the simplest improvements over flat shading is *Gouraud shading* (also called *intensity interpolation*) **[Gour71]**. In the Gouraud shading method, the intensity (or colour components in the case of colour images) at any pixel is linearly interpolated from the intensity at the vertices. The main advantage of Gouraud shading is that the colour gradient is constant over the surface allowing the interpolation to be implemented using simple addition (in a similar manner to that used for the edge positions in the span generation). Other, potentially more efficient solutions are described in section **A.3**.

Figure 3.11 shows the same object as above, rendered with Gouraud shading. The edges of faces are a lot less pronounced. An effect called *Mach banding* occurs which causes some of the edges to remain particularly noticeable; this is an optical illusion caused by the human eye's sensitivity to changes in shading. This causes apparent brightening of edges where the intensity gradient changes.

3.14.3 Hidden surface removal

It is necessary to determine at each pixel which of the primitives overlapping that pixel is closest to the viewer and hence visible. Many hidden surface removal techniques exist, often operating at various different levels in the pipeline. The technique described

Figure 3.11: A Gouraud shaded teapot

here is the z-buffer which is used in most commercially available rendering systems. Its main advantage is that it is extremely simple to implement.

In the z-buffer algorithm another section is added to the frame buffer which holds a measure of the depth of the currently visible surface for every pixel on the display. A simple measure of depth is the reciprocal of the Z coordinate[1]. Calculating the depth value at each pixel is performed by linearly interpolating in exactly the same way as the intensities are interpolated. At each pixel the newly calculated depth value is compared to the stored value. If the new pixel is in front of the stored pixel, then the new pixel value (complete with its calculated depth and colour) is written into the frame buffer. A simple implementation of this algorithm requires a read-compare-write operation which would more than halve the speed of rendering. An optimisation is to store the depth buffer in VRAM and to read the old depth values out of the port normally used to drive the digital to analogue converter **[GGSS89]**. In systems supporting z-buffering and Gouraud shading, the span generators and span fillers contain a large number of similar interpolators. An alternative solution (which sacrifices speed for a lower cost) is to have

1. Due to the effects of perspective, the raw Z value would not be suitable for linear interpolation. An explanation of why the reciprocal of Z (the proximity) can be used is given in **[GrLi87].**

one interpolator and to store the current values in a small block of RAM. The parameters are then updated one at a time. Similarly it is possible to share the interpolators between span generation and span filling.

3.15 Hardware versus software

The algorithms presented in this chapter (particularly for rasterisation) are suited to hardware implementations and may not be suitable for software implementation. In particular the z-buffer algorithm would incur a large amount of computation at each pixel (read, calculate depth, compare, write depth). **Appendix A** contains a description of a number of other hidden surface removal algorithms.

In general, it is easier to implement in hardware an algorithm which performs many very simple identical operations (like the z-buffer algorithm) and it is easier to implement in software algorithms which perform complex or widely varying operations. The microcoded processors sometimes used for the geometry subsystem represent a halfway point between software and hardware.