Chapter 5

Rendering complexity

This chapter describes ways of measuring the complexity involved in rendering scenes, the source of the complexity, and some ways of reducing it. Some of these complexity metrics are applied to a number of applications, including some of those described in section **1.2**.

5.1 Complexity metrics

This section examines a number of complexity metrics which can be used to judge how difficult it is to render a scene, measure the capabilities of rendering hardware and measure the needs of VR applications. Constructing useful measures is difficult since a task that is difficult on one rendering system, may be easy on another. Another problem to be considered when choosing a measure, is that the way in which a scene is processed by one stage of the rendering pipeline affects the complexity of processing performed by later stages. A number of different complexity metrics will be considered:

- Model complexity - Total number of primitives in the model

  In the conventional rendering system shown in **Chapter 3** , the number of primitives in the model will affect the amount of processing required in all levels of the pipeline. Some applications use the same object many times (for example trees in a forest); one definition can be *instantiated* many times rather than storing multiple copies. Each instance needs only a transformation matrix defining the its position, orientation and size. Instantiation can also help reduce the large bandwidth required between host and rendering system in immediate mode systems.

- Complexity of the primitives

  Complex primitive descriptions (such as curved surfaces) usually take longer to render than polygons, however one complex primitive may be able to replace many hundreds of polygons.

  "It may become appropriate to think of systems that can render more complex primitives. Pretty soon we will have polygon rendering systems that can do Gouraud and Phong shaded polygons. However, at a point in time, data bases will become so large that polygons are in the order of the size of a pixel. At that point using polygons makes no sense any more", John Eyles, in a discussion on subsystems in **[KuSt88]**

  As pointed out in the extract above and the figures given in section **5.2**, the size of the primitives used is becoming impracticably small. These small sizes cause the setup and transformation times to dominate the rendering time. Some rendering systems provide microcoded support for the tessellation of curved surfaces but very few render curved surfaces directly. One such technique is described in **[LiSP87]**; which shows that the technique is more efficient than subdivision methods. Conventional VRAM allows horizontally adjacent pixels to be plotted faster than arbitrary pixels; unfortunately, the filling of short spans present in small primitives, and also the technique presented in **[LiSP87]** do not take advantage of fast adjacent pixel plotting, thus reducing the available frame buffer bandwidth. Advanced memory architectures such as FBRAM (described in **[DeSL94]** and section **6.3.2**) - which include caching and ALU operations inside the frame-buffer memory - reduce the penalty for random access to the frame buffer and may therefore allow these surface plotting algorithms to be efficient.

  In a conventional rendering system, the time taken to rasterise one polygon is proportional to its screen area, however many scenes consist mainly of small primitives. It is tempting to argue that since many primitives are small, hardware designed for filling large areas is no longer effective; in practice primitives are found in a wide variety of sizes, some of which will still benefit from the filling hardware. In **[DWSD88]** it is argued that although the majority of triangles in a simple test scene are very small (81% were below 10 pixels in size) a large proportion of the frame buffer accesses are due to a few larger triangles. The figures given, indicate that 40% of the frame buffer accesses were due to the 1% of the triangles larger than 100 pixels; this indicates that hardware filling accelerators are still useful.

Although the time to render a primitive depends on its size in most architectures, a few can render primitives in constant time; in particular Pixel-Planes 4 **[FGHS85]**, described in detail in section **6.1.7.2**, renders any triangle in a fixed time. Although the size of primitives is irrelevant, a scene consisting of lots of small primitives will still take longer than a scene containing a few large primitives. Another interesting property of Pixel-Planes 4 is that it can render spheres (with some limitations) in constant time, and at a similar speed to triangles; this example shows that while a sphere is a highly complex object on one rendering system it can be regarded as a primitive on another.

- Shading considerations

  A simple flat shaded triangle can be rendered in less time than a Gouraud shaded triangle. Similarly Phong shading (described in section **5.4.3**) increases the rasterisation and setup times considerably - however it shows surface highlights that Gouraud shading will lose. Another algorithm which renders highlights effectively is the *Highlight shading* algorithm described in **[PöHo89]**. This subdivides primitives finely around highlights and Gouraud shades each of the resulting primitives. Less subdivision is used on sections of the primitive which do not contain highlights.

- Scene complexity

  The scene complexity is the number of primitives (or pixels, edges or vertices) which are in view; it can vary greatly depending on the position and orientation of the viewer. Provided that the trivial reject/accept block of the rendering pipeline has been effective, the scene complexity corresponds closely to the amount of rendering work. The time complexity of a Z buffer rasterisation and hidden surface algorithm is linear with the number of pixels rendered and is thus related to the scene complexity. With other hidden surface algorithms, this may depend on the number of edges, or a range of other properties of the scene. It is therefore necessary when comparing systems to be careful about the measure of scene complexity used.

- Depth complexity

  The depth complexity is the average number of primitives which cover a pixel on the screen. Systems which display few objects have low depth complexity, however the depth complexity of views of complex buildings or a city can be very high. The simple Z buffer algorithm renders every primitive in view, thus a system rendering a scene where a viewer is standing outside a large building, would have to render the building, every room in the building, every desk in the building and every object in each desk; even though they are all hidden inside the building.

- Vertices in the scene

  The number of vertices that need to be transformed obviously influences the load on the geometry subsystem. This measure is not necessarily related directly to the number of primitives in the scene, since some representations (such as triangle strips) share vertices between primitives and thus reduce the amount of transformation necessary.

- Number of nodes in the hierarchy

  Traversing the hierarchical model definition can be expensive; each time the algorithm descends a branch of the hierarchy one or more matrix multiplications are required.

- Stability of the model

  Some rendering algorithms benefit from static models - it is therefore useful to measure how much and how often the scene changes.

These measures are not independent of each other - for example it is obvious that a complex model is likely to generate views with a high scene complexity. Some of the most useful measures for comparing different applications are the scene and model complexities. Models can grow to be extremely complex and contain hundreds of thousands of primitives and it is thus necessary to discover the sources of this complexity:

- Tessellated objects - curved surfaces which may be tessellated into many thousands of polygons.
- Repeated objects - for example trees in a forest or windows on a large building
- Highly complex objects - for example polygonal representations of voxel data sets (see section **A.7.3**)
- Large scenes - VR applications often try and model very large areas such as an entire city; a representation of every object in such a scene would generate a very complex model.
- Large objects - Terrain models are especially large and complex. **[Prat93]** states that it is common to use a terrain model representing around fifty square kilometres and that it is necessary to store this in a fairly high level of detail.
- Surface subdivision for lighting - radiosity and highlight shading both subdivide surfaces to improve the representation of lighting effects; this reduces the complexity of each primitive (since they can be Gouraud shaded) but generates many primitives.

5.2 The complexity of some VR applications

This section examines some of the applications presented in section **1.2** and presents complexity figures for them. A summary of the complexity information is given in table 5.1. Seven models have been analysed:

- The CADCentre 'test' model

  This model, shown in Figure 5.1, has relatively few primitives. The view given here shows the whole model. It is characterised by the fact that a large portion of the model is hidden by a low wall.

Figure 5.1: The CADCentre test model

- The CADCentre 'plant' model

  This model is a lot more complex than the preceding model. As can be seen from Figure 5.2 it is characterised by open spaces, some large occluding cylinders and by an office block to the left of the image. Figure 5.3 shows a depth complexity map for this image (darker areas indicate more pixel writes have occurred at that position); note the high depth complexity in the office block area and that the mean triangle size is very small at 16.2 pixels. The CADCentre system which is currently available to the Advanced Interfaces Group does not allow the model to be changed. The long term aim is however to produce a VR system for the designers of these models.

Figure 5.2: The CADCentre plant model

	CADCentre test model	CADCentre plant	Division house	Division bedroom	Jack	Radiosity scene
Triangles at rasterisation stage	4579	22952	2455	506	9439	4612
Mean depth complexity	1.33	3.06	1.67	2.65	2.35	1.56
Mean triangle area	31.1	16.2	179.5	1436.9	45.1	88.0
Number of vertices	18948	92195	16340	4585	18944	17024
Display size	423 x 291	423 x 344	601 x 466	601 x 466	601 x 333	598 x 481

Table 5.1: Complexity of VR applications

Figure 5.3: Depth complexity map of the CADCentre plant model

- A house from the Division dVs system

  This model was provided with a demonstration version of the Division dVs system. The house model has many rooms, each containing furniture. The view which was analysed (shown in Figure 5.4) is from the outside, observing the whole house. The house is drawn on a backdrop which effectively increases the depth complexity by one. Many of the objects inside the house are also drawn. The dVs system allows the user to move objects about and open doors and drawers, thus the model can change quite dramatically.

- A bedroom inside the Division house

  This model (shown in Figure 5.5) shows the effectiveness of a culling algorithm used in the dVs system. Only the primitives needed for a particular room are drawn and thus the rest of the house model is culled out. The primitives in this scene are unusually large and the depth complexity is high. It is suspected that this is because the backdrop of the outside world is drawn which covers

Figure 5.4: The Division house model

most of the image (and consists of large polygons) and on top of that walls cover most of the image. Both of these account for a depth complexity of one each, the rest of the depth complexity being due to objects in the room.

Figure 5.5: A bedroom in the Division dVs house.

- Jack

  This model is a scene from the *Jack* application described in section **1.2**. A large amount of detail in this scene is spent on the dashboard and tessellating the cylindrical primitives making up *Jack*. This model is shown in Figure 5.6. The user can cause *Jack* to move and can load other models for *Jack* to interact with.

Figure 5.6: A scene from the *Jack* package.

- A scene lit using radiosity

  The illumination in this model was calculated using a radiosity algorithm which is often used to light scenes for architectural walk-through applications. It is characterised by a large number of small patches making up all the surfaces. The scene is shown in Figure 5.7. The scene cannot be modified in real time. The radiosity technique is described in section **A.7.2**.

5.2.1 The Notes on statistics

This section describes the statistics in table 5.1. All the applications were monitored using the GLprof profiling system and the rasterisation process was simulated in the CSim simulator. These systems are described in detail in **Appendix B** .

Figure 5.7: A scene lit using a radiosity algorithm

- Triangles at rasterisation stage

  The figures presented in this line are the total number of triangles presented to the CSim simulator by the modified OpenGL X server. The triangles have already been culled and clipped.

- Mean depth complexity

  This figure is the total number of writes to the frame buffer divided by the area that the model occupied.

- Mean triangle area

  The area of the triangles as passed to the CSim simulator. The area is calculated from the raw vertex coordinates and does not reflect the fact that each triangle will occupy an integral number of pixels in the frame buffer.

- Number of vertices

  This figure is a count of the number of calls that the application made to the graphics library routines which specify a vertex position (such as the v3f() gl library call). This was produced by analysing the *GLprof* trace files and it gives a measure of the amount of work required in the transformation subsystem.

- Display size

  The area which the model occupied in the frame buffer. This is taken as the difference between the minimum and maximum coordinate values on each axis.

Each application used different primitives and model storage methods; this makes it difficult to obtain a useful measure of the number of primitives and/or objects in the entire model.

5.3 Efficient use of the hardware

It is not possible to determine the rendering time of a scene purely from the metrics given above and details of the rendering hardware; many other factors affect the behaviour of the rendering system. One such factor is the order of rendering.

Figure 5.8 shows a graph of triangle sizes in a typical scene, in the order in which they were rendered. It can be seen from this graph that primitives of similar sizes seem to group together. One reason for this is that an application is likely to render all the primitives for a particular object at the same time, and that they are likely to be similar in size. For example when rendering a room, an application may render all the walls and then the objects inside the room, one by one. The walls, which would be sent close together, are likely to consist of primitives much larger than the objects in the room. When a number of large primitives are rendered sequentially, the span generator and filler will be heavily loaded, and even moderately sized FIFOs will not be able to stop the geometry processor stalling. For this reason it would be helpful if the rendering of a number of objects of different sizes, could be interleaved, thus producing a mixture of primitive sizes. Both the hierarchical Z buffer and the new culling algorithm described in section **5.6** operate most efficiently if large objects, likely to obscure much of the rest of the scene are rendered first. This is a different (and probably contradictory) constraint on the order or rendering..

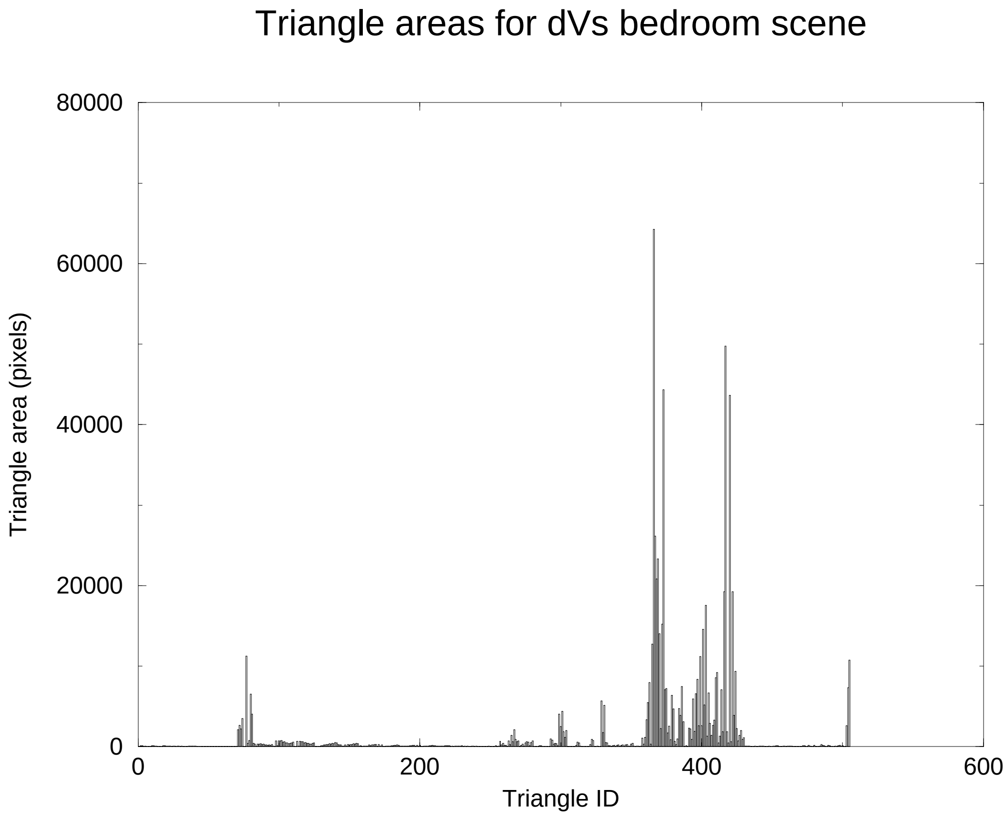

Figure 5.8: Triangle areas in rendered order

5.4 Other complexity issues

Some rendering techniques which affect the quality of the rendered image can increase the complexity of the rendering task. However the fact that the rendering is of a higher quality can mean that a simpler model can be used or that the existing one is more effective. This section discusses some of these techniques.

5.4.1 Texture

The ability to map an image onto a surface is called *texture mapping*. It is useful for the following reasons:

- Replacing geometry

  In a model of an office block each window could be represented as a large number of small polygons. These polygons can dominate the time taken to render the building. An alternative solution, which uses only a few polygons, is to represent the office block as a cuboid and repeatedly map an image of a window onto its sides. As long as the image is of a high enough resolution, or the office block is seen from a suitably large distance, this can be convincing.

Another technique for replacing geometry is called *billboarding*; often used for symmetric objects. An image of the object is rendered and texture mapped onto a single polygon which is constantly rotated to face the viewer. Trees can be simulated fairly well with this technique (when seen from a distance) **[RoHe94]**.

- Simulating material properties

  Texture mapping can be used to simulate the appearance of wood, stone and other patterned materials. Simple two dimensional texture mapping (where an image is used) is reasonably effective for a flat surface (like the top of a desk) but the effect is unconvincing on carved objects. An extension, called *solid texture*, holds a three dimensional texture, allowing realistic carved wood and stone effects to be created **[Perl85]**. Simulated materials can improve the realism of an image greatly, at little cost to the geometry section of the pipeline.

- Providing visual cues

  Under uniform lighting conditions it can be difficult to tell whether a plane is slanting and even more difficult to judge its angle. Wanger et. al. **[WaFG92]** show various examples of texture removing this type of ambiguity. Figure 5.9 shows two images from the NPSNET system **[Prat93]** one with and one without texture. The hill in the background looks flat in the scene without texturing, but not in the textured scene. The use of texture in NPSNET also helps the perception of movement since the rate at which the texture moves helps the viewer to judge his speed.

(a)

(b)

Figure 5.9: NPSNET scene showing the effect of texture

- Two dimensional images

  In many scenes there are often cases where two dimensional objects (signs, text on a page, and the markings on a road) must be displayed. These are easily simulated by the use of texture mapping.

- Window views

  Sometimes there is the need to simulate the view out of a window. If the user is constrained to be inside a building it is wasteful to generate and render the whole world just so he can look out at it. An alternative is to render a view and then texture map it onto the window. This does not look completely correct due to the lack of parallax cues but is sufficient for many applications. A limited amount of parallax can be achieved by texture mapping the view onto a plane placed some distance behind the window.

Texture mapping offers many advantages but to perform it correctly is expensive. Low cost implementations of texture mapping suffer from perspective distortion and aliasing; both of which can be particularly objectionable in moving scenes. Simple forms of texture mapping are now appearing in low cost entertainment software; however this software tends to limit the viewing parameters to simplify traversal through the texture map and the work required to perform simple perspective correction. It is unusual for this type of software to perform antialiasing on the texture maps.

5.4.2 Shadows

Figure 5.10(a) shows an image containing two spheres; the sphere on the right is larger, further away and lower down; however it is impossible for the viewer to tell this. Figure 5.10(b) is the same scene with shadows added, it is now clear that the right sphere is further away and lower down.[1] The fact that it occupies the same space on the display indicates it must also be larger. In **[WaFG92]** the effectiveness of various enhancements including shadows was tested. These tests showed that the inclusion of shadows produced a large increase in positional and size accuracy in a number of tasks performed by volunteers.

There exist a number of different methods of generating shadows, a few of which are described briefly here:

- Recursive ray tracing

  Shadow generation may easily be added to the ray tracing algorithm which is described in section **A.7.1**.

1. These images were generated using the SIPP package written by Jonas Yngvesson and Inge Wallin. The images are based on figure 9 of **[WaFG92]**.

(a)

(b)

Figure 5.10: A scene with and without shadows

- Shadow volumes

  The shadow volume technique (described in **[Crow77]**) constructs a number of *shadow polygons* which enclose an area which is in shadow. These polygons are then rendered using a modified scan conversion algorithm which reduces the intensity of any polygon which is enclosed by a pair of shadow polygons. This method has been implemented on many types of rendering system including scan line based systems and on Pixel-Planes.

- Depth Maps

  The depth map algorithm is a two pass algorithm. In the first pass the scene is rendered using each light position as the viewpoint; the resultant Z buffers are then used as depth maps. These contain the distance to the closest object, and thus the object illuminated by the light, at each pixel. The coordinate space as seen from the lights position is known as *light space*. The second pass is an enhanced conventional rendering algorithm. As each pixel is being plotted its position is transformed into light space and the corresponding depth map entry retrieved. If the new pixel is further from the light than the entry in the depth map for that light, that pixel is in shadow and its intensity is reduced. Depth maps are large, suffer from aliasing and require a lot of processing to transform points from screen space to light space. One such algorithm (which removes some of the aliasing problems) is described in **[ReSC87]**.

All of these techniques are relatively expensive however there are a number of other methods which can reduce the amount of computation required. In many applications the designer knows that it is only important for certain objects to generate shadows and for only a few objects to have shadows cast upon them. For example in a flight simulator it may be acceptable for shadows to be generated only by aircraft and only cast onto the ground.

5.4.3 Phong shading

The simple linear interpolation of intensity used in Gouraud shading suffers from a number of problems. Small highlights can be lost if they fall completely inside a polygon and the effect of highlights which fall on vertices is greatly diminished. These problems lead to surfaces appearing dull. In the Phong shading method a full lighting model is evaluated at each pixel. This produces much more realistic highlights but incurs a much higher computational cost. This cost (which includes a division and square root at each pixel **[BiWe86]**) is too expensive for most applications. Many approximations have been made to the Phong shading model including *Fast Phong Shading* by Bishop and Weimer **[BiWe86]**. This technique reduces the amount of computation per pixel to five additions and a table look-up.

For many applications Gouraud shading is quite sufficient and it is therefore questionable whether it is worth implementing Phong shading. Where previously high levels of tessellation were used to generate surfaces which demonstrated effective highlights, Phong shading effectively reduces the number of primitives required, however where tessellation was heavily used to produce clean intersections between objects and clean silhouette images, Phong shading will provide no benefit. An example of a system which provides Phong shading is described in **[Clau92]**; Ute Claussen who described that system was concerned whether the extra hardware was justifiable:

> *"Furthermore, it is questionable if the amount of hardware is justifiable if preprocessing methods like the highlight shading method could lead to a similar quality without the need for any shading stage in the system.",* **[Clau92]**.

5.4.4 Transparency

Some applications require the representation of partially transparent surfaces; these may be used to represent surfaces which are transparent in reality (such as windows) or they may be used to allow the detail inside an object to be seen while still displaying its outer surface.

A simple way of incorporating transparency in existing systems is through the use of alpha blending hardware originally intended for image composition **[PoDu84]**. In this technique each polygon (and subsequently each pixel) is assigned an *alpha* value. When a pixel is to be written into the frame buffer the previous colour and alpha information stored for that pixel in the frame buffer is read and combined with the new colour. The alpha value (in this context) acts as a measure of the opacity of the polygon; an alpha value of one represents a completely opaque polygon and a value of zero represents a completely transparent polygon. Although this technique is simple (and often effective) it is difficult to integrate with the Z buffered rendering algorithm; the main problem being that the process is highly order dependent. Consider four polygons, two partially transparent and two opaque arranged so that they are ordered transparent, opaque, transparent, opaque in increasing distance from the viewer (Figure 5.11). For the purpose of description these will be called A,B,C and D where A is nearest to the eye and D furthest. If they are rendered in the order A,B,C,D using the process described above, B,C and D would not get rendered since the Z values in the depth buffer would have been updated by the process of rendering A; no image would be generated since the only primitive to be rendered (A) is transparent. If they were rendered in the reverse order the correct image would be formed, with C modifying the appearance of D, B obscuring that image and A modifying the appearance of B. Some graphics libraries reorder requests for transparent surfaces so that the correct image is generated.

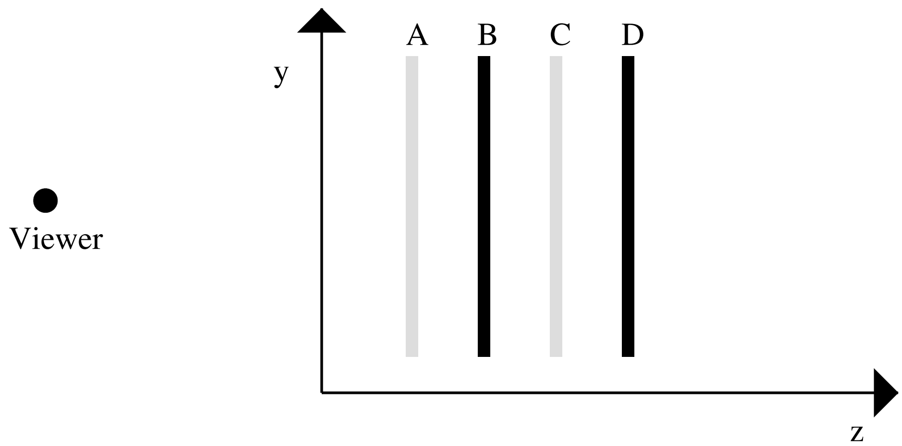

Figure 5.11: A mixture of transparent and opaque primitives

Primitives A and C are transparent; B and D are opaque.

Another common transparency method is *screen door transparency* which ignores some of the pixels in transparent primitives completely thus leaving tiny ‘holes’ in the primitive. At low resolutions the effect is obvious and potentially disturbing, while even at high resolutions it is possible for a number of overlapping transparent primitives to obscure other objects if the holes in one primitive completely align with the rendered parts of another.

Very few techniques handle transparency without ordering problems and even less handle the refractive behaviour of transparent objects. Recursive ray tracing handles transparency (with refraction) and is described in section **A.7.1**. The only piece of rendering hardware to handle transparency correctly known to the author is the experimental system described in section **6.1.6**.

5.5 Reducing complexity

This section looks at ways of reducing the amount of work the rendering pipeline has to do. Complexity reduction is only worth performing if the reduction process is less computationally expensive than the saving made or if the process moves work from a highly loaded section of the pipeline to a lightly loaded section.

5.5.1 Detail Elision

When creating a model of an object it is normally necessary to include enough detail so that the object looks convincing even when viewed at a small distance. However if the object is far away it is wasteful to use a highly detailed representation. Switching between different representations is called *Detail Elision* - or simply *Level of detail switching*. Different models of an object are used depending on the size of the object and the time available to render it. The detail in a scene can then be tailored to a desired frame rate and, using prediction techniques, the frame rate can be kept more uniform. Funkhouser and Séquin **[FuSé93]** point out that there are some scenes where a large number of objects are inevitably visible - they use the examples of a fully stocked shop, a complete aeroplane engine and a library - in these examples detail elision is essential to provide a satisfactory redraw rate. To provide level of detail switching three separate problems must be solved:

- Creating object descriptions at various levels of detail
- Choosing which level of detail to use for each object
- Ensuring that changes in detail level are not disturbing to the user

Creation of objects at many levels of detail

Each object must have a number of representations which have a different combination of quality and rendering cost; objects which are manually-designed usually require that the designer also defines less detailed versions of the objects. DeHaemer and Zyda describe methods of producing reduced complexity descriptions of models generated by three dimensional digitising equipment (such as those produced by Cyberware Laboratories of Monterey, California) **[DeZy91]**. The techniques described in that paper require much processing and therefore it is necessary to compute and store all the representations of the object. Other, less accurate, techniques - such as missing one in every ‘n’ points out, can be done during rendering and thus have lower storage requirements. For objects that are tessellated by the rendering system this can be achieved by altering the parameters of the tessellation process. One difficulty when applying this technique to a conventional system is that the tessellation stage is independent of the viewing position and thus it is not possible to relate the size of the object in the user’s view to the amount of tessellation. Often this leads to the use of inappropriate levels of tessellation. A solution to this is to tessellate further down the pipeline; this can also lead to a reduction in the load on the modelling transformation engine. Most of the techniques discussed in this section have considered reducing the amount of geometry in each object’s description; it is also possible to select different rendering techniques (i.e. shading, texturing etc.) and then to use various combinations of the geometric descriptions and rendering techniques.

Choosing which level of detail to use for each object

One of the simplest methods of selecting the model version is to base the selection purely on the area of the object on the display. Each version of the model has a ‘maximum area’ value associated with it; simple representations have small maximum area values indicating they are only suitable when viewed from a long distance. The representation selected is the simplest which has a maximum area value greater or equal to the area of the object on the display.

A problem with this technique is that the level of detail of each object is selected individually and there is thus no control over the time taken to render the entire frame. A technique used in some flight simulators is to alter the thresholds for the level of detail selection from frame to frame based on the time that the previous frame took to render. Thus if the previous frame took too long, the level of detail is reduced in subsequent frames; this is a *reactive* system. Funkhouser and Séquin in **[FuSé93]** state that such systems are not suitable for VR environments since the scene complexity can change rapidly from frame to frame, which can cause over compensation and oscillation in the level of detail selection system. They describe a *predictive* system which attempts to work out the detail that can be afforded to the *current* frame. The aim of their method is to produce the best possible image at a given frame rate. They define heuristics which produce a measure of the rendering difficulty and the benefit to the quality of the image for every combination of object, level of detail and rendering technique. They then maximise the sum of the benefits of all the objects in the scene while ensuring that the sum of the costs is at or below the capabilities of the hardware. This optimisation takes between 10ms and 30ms to compute; this would be a problem if this time became part of the overall latency of the system. It is however important to note that this extra computation helped reduce the rendering time significantly (when compared to a system with no detail elision) and produced a considerably more uniform rendering time.

Ensuring that changes in detail level are not disturbing to the user

Users are often very sensitive to sudden changes in the representation of objects. It is therefore important that changes are hidden by blending between the two models (by using transparency blending or interpolation techniques). Ensuring that the different representations have a similar appearance helps to reduce problems caused by switching representations. It is also important to use hysteresis in deciding which representation to use. This avoids the problem of the system cycling rapidly between representations in marginal cases.

5.5.2 Culling

Culling is used here to describe techniques which determine sections of the model that are not visible (because they are hidden by other objects or are out of the field of view), and then remove them from the display process. The techniques used in the trivial accept/reject stage of the conventional rendering pipeline described in **Chapter 3** are examples of culling algorithms. Culling algorithms do not have to remove all hidden polygons; they must just simplify the model for later stages.

One problem with culling algorithms in general is that they are often highly unpredictable. This breaks the aim of a constant redraw rate stated in **Chapter 4** . Because of this lack of predictability, if a guaranteed redraw rate is required, the rendering system must be configured with enough processing power to cope if no culling is performed. Culling may then be pointless, since the system could have rendered the scene within the required frame time without culling. In practice culling is useful since in many cases, systems do not have sufficient processing power to handle the worst case. To be able to meet the constant redraw rate requirements culling algorithms which are more predictable are needed.

5.5.2.1 Space subdivision for view frustum culling

The technique described in **Chapter 3** for culling against the viewing frustum is wasteful, since by the time the primitives get to the trivial accept/reject phase a large amount of processing has already been carried out on them. It would be better if the database traversal could be carried out selectively and only output primitives in or close to the viewing frustum. One way of doing this is to preprocess the scene and create another data structure whose structure reflects the spatial organisation of the objects in the database. Note that it is common to treat whole objects at once rather than individual primitives; this reduces the amount of processing and storage required. A number of different techniques exist based around different data structures:

- Grid cell enumeration

  The whole of the world space is split up into a three dimensional array of cells. Each cell has a list of objects which occupy the space in the world that the cell represents. The placement of objects in cells is a preprocessing step - which may take a significant amount of time for a large model. When a frame is to be rendered every cell which is in the view frustum is visited and all objects on the cells' object list are rendered.

  A heuristic approach is needed to determine the size of grid cell to use; if the cells are too large then the benefit of using the technique will be reduced, if the cells are too small the time to traverse the array starts to become unacceptable.

  Grid cell techniques have a number of disadvantages. The first is that the grid cell array uses very large amounts of memory, the second disadvantage is that time will be spent traversing empty cells. The third disadvantage is that even distribution of the cells throughout the world, tends to create cases in which there will exist many empty cells and a few cells with a large number of objects.

- Octree encoding

  Octree encoding is a different method of subdividing space which does not suffer from the empty cells and uneven distribution of the grid cell technique. The technique involves splitting the world space half way along each axis (Figure 5.12). This creates eight cells, which can then be similarly subdivided to create a hierarchy. The octree is stored as a tree, where each node has eight sub-trees (representing each of its child cells); objects are stored at the nodes of the tree. If a cell contains a large number of objects it may be subdivided and the objects redistributed into the new cells. When it is time to render a frame the tree is traversed. Any sub-tree which is completely outside of the viewing frustum is not considered.

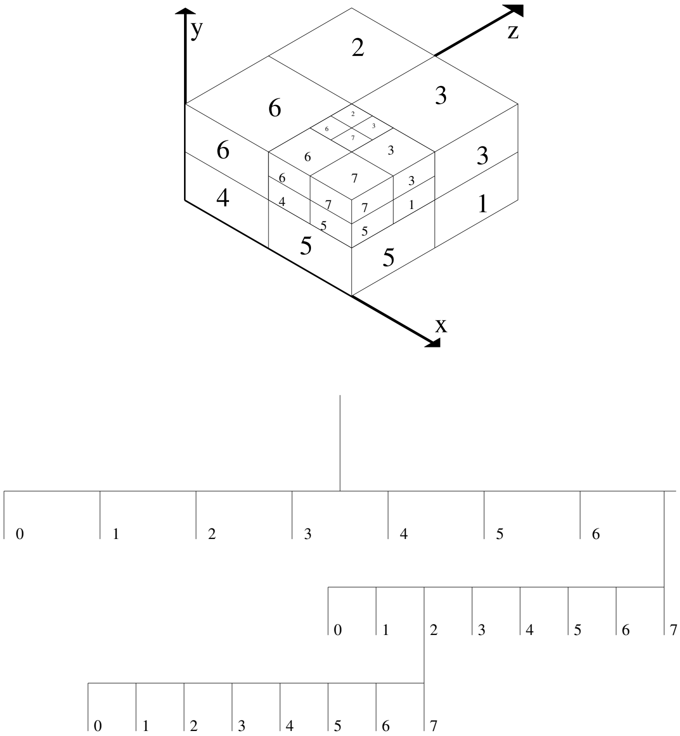

Figure 5.12: Octree encoding

The top diagram shows a cube subdivided by an octree, the bottom diagram shows the tree representation of the same subdivision.

- Potentially visible sets

An approach often used in architectural walk-through applications, where there are easily identifiable rooms is to create *Potentially Visible Sets* (PVS) **[AiRB90]**. The world space is again subdivided, this time into cells and each cell has associated with it a set of objects which are potentially visible from somewhere in that cell. To render an image it is first necessary to find out which cell the viewer is in; then all objects in the PVS for that cell can be rendered. The cells tend to correspond to rooms in a building. Two main problems exist with this method. The first problem is doorways and other openings which can cause the size of the potentially visible set to increase and change (when a door is opened), the other problem is that the potentially visible sets can take a long time to create, and produce very large data structures **[GrKM93]**.

5.5.2.2 Hierarchical Z Buffer

The hierarchical Z buffer **[GrKM93]**,is designed for scenes with a large depth complexity - such as large buildings. Its aim is to cull out objects which are hidden by walls or other occluders early in the rendering process. It uses three data structures: an object-space octree, an Image-space Z pyramid and a temporal coherence list. The object-space octree is constructed in a similar way to the octree described above. Greene, Kass and Miller **[GrKM93]** note that if a cube of an octree is completely hidden then everything in that cube is hidden. To determine this, it is necessary to scan convert the cube (without modifying the frame buffer). If all of the pixels representing the cube are hidden (as determined by the Z buffer test) then it is not necessary to descend into that branch of the octree. This test can be integrated in the standard procedure for rendering objects in an octree. This method exhibits *object space coherence*, which is rare in Z buffer based rendering methods. An algorithm which uses object space coherence uses one test to discover the visibility of groups of close objects, instead of performing a test on each one. All types of coherence are useful in reducing the amount of computation needed during rendering.

There exist two main problems with this octree technique. The first is that if a primitive straddles one of the dividing planes of the root octree then it will always get rendered, similarly an object straddling a plane of an octree anywhere in the tree is more likely to get rendered than the same primitive translated inside the octree. The other problem is that the scan conversion of the octree cubes is expensive.

Their solution to this second problem is to create an *Image-space Z pyramid*. This uses image space coherence - that is one test is used to determine the visibility of a *group of pixels* rather than individual pixels. The Image-space Z pyramid consists of a number of Z buffers - each a quarter of the size of the next finer buffer. It is possible to test an object for visibility against this hierarchical Z buffer by starting at the coarsest level of the pyramid in which the object fits into a single pixel, and comparing the nearest Z on the test object with the value in the buffer. If the nearest Z of the object is behind the buffer value, then it is possible to say definitely that the object is hidden; otherwise the test descends to the next finer level of the buffer. At each stage the Z buffer value is compared to the nearest Z of the object *in that cell*; the process involved in calculating this is expensive. An alternative suggested in **[GrKM93]** is to use the nearest Z of the object in all tests; this modifications means that the algorithm cannot prove that an object is visible, only that it is potentially visible or definitely hidden. When updating the Z buffer (when plotting objects) it is necessary to propagate new values back up the pyramid until there is no change in the value already stored.

The order of rendering is important in determining how effective culling is. If large obstructing objects are processed earlier they are likely to help cull a large number of objects later. One way of helping to achieve this is to render the cubes of the octree closest to the viewer first. The method used in **[GrKM93]** is to maintain a list of visible octree cubes from the previous frame - this is the *temporal coherence list*. Temporal coherence relies on the scene and the view changing little from one frame to the next. This method ensures that many of the large occluding objects are rendered using a simplified algorithm before the more expensive hierarchical system is used, and the remaining cubes will be culled out of the Z buffer using less recursion since many of the occluders are already present.

Greene, Kass and Miller obtain large speed increases using a software implementation of this technique on very complex models. They look at the possibility of using existing graphics hardware and complain that many systems do not provide a means of testing for visibility and those that do take many milliseconds to return an answer. This delay is inevitable due to the pipelining of the rendering system.

In the author's opinion implementing the hierarchical Z buffer in hardware efficiently would be problematic:

- Either a large number of separate Z buffers would be needed or a cleverly addressed single Z buffer would be required. With a 1024 x 1024 display eleven separate buffers are required.
- A normal pixel plot operation is now considerably more complex. An unknown number of buffer writes is required at each pixel - each with comparisons to see if the next buffer needs modification. This requires either a lot of parallel hardware or a long chain of operations.

5.6 A new simple culling algorithm

Having described existing culling techniques, this section describes a new technique which the author has developed. The concerns which have caused its developments are:

- A dislike of secondary data structures

  All the techniques described above require another data structure which is not related to the hierarchical model description. These data structures take time to create, require more memory and make modification more complex. The fact that they have little relation to the structure of the model mean that a simple operation on the model leads to a lengthy procedure often involving the modification of many sections of the culling data structure.

- The need for hardware acceleration

  The host processor has enough work to handle - in complex environments the host is already burdened with collision detection, input handling, dynamic simulation and many other housekeeping duties - it is undesirable to load it further. To provide hardware acceleration it is desirable that the culling technique be as simple as possible, to allow it to be implemented in specialist hardware or in microcode.

A number of related techniques will be described; simulations have been carried out on the simplest techniques and are described below. All techniques are designed to be used with a simple extension of the existing Z buffer technique.

5.6.1 Polygonal occlusion

Given two convex polygons A and B, A occludes B if all vertices of B project onto the same section of the viewing plane as A and all those vertices of B are behind A at those points (Figure 5.13 shows this situation with a view of the yz plane). If these were the

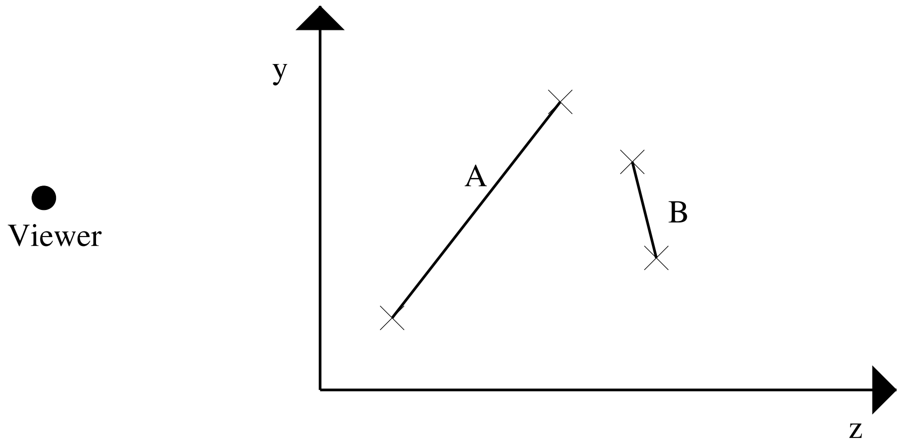

Figure 5.13: Simple polygonal occlusion

The view of B is obscured by A since both vertices of B are obscured by A.

only polygons in the scene, it would be possible to carry out this simple test while plotting B by examining the existing values in the Z buffer at the projected vertices of B and comparing them with the Z value of these vertices. Unfortunately a scene with just two polygons is probably useless and with more polygons the scheme breaks down

(Figure 5.14). This problem is removed by adding a *Polygon ID buffer* which for every

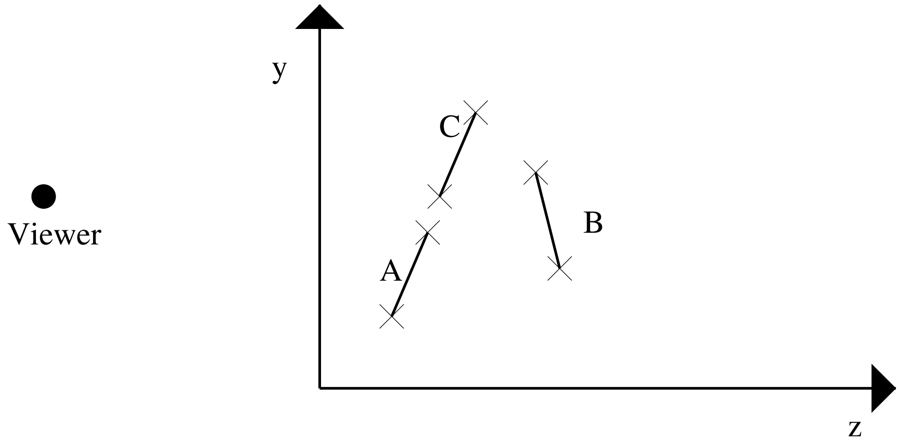

Figure 5.14: Polygonal occlusion - a more complex case

B is still visible even though both of its vertices are obscured.

pixel on the display stores a unique polygon identification (ID). At the start of the frame the buffer is cleared to an easily identifiable code to indicate that no polygon is occupying that pixel. As a pixel is plotted and passes the Z buffer test the old ID is replaced by the ID of the new polygon. This is a simple extension to the conventional rendering algorithm and (besides the cost of the buffer itself) is inexpensive to implement and should have little effect on rendering speed.

This culling algorithm tests the polygon ID at every vertex of the polygon which is about to be rendered. If the ID is not the same at all corners, or the ID is equal to the 'no polygon present' code the new polygon is rendered anyway. If the polygon IDs are all identical then a comparison of the Z buffer values with the polygon vertex Z values is carried out. If all new vertices are behind the currently stored pixels then the polygon can be culled since it is completely hidden. This algorithm is implemented in the pipeline between the geometry and rasterisation stages. It is not necessary to wait for the previous primitive to finish rasterisation before carrying out the cull test; if the previous primitive would have occluded the new primitive the new primitive will pass through to the rasterisation stage - it will not produce an incorrect image but will reduce the effectiveness of the cull a little (while ensuring that the pipeline does not stall). The only operations involved in the culling test are memory accesses and comparisons, which can be implemented in dedicated hardware or can be programmed on the processor used to carry out the splitting of triangles into trapeziums. The only complication (which may be severe) is that it is necessary to arbitrate between accesses to the Z buffer and ID memory by the rasterisation engine and the culling hardware. A block diagram is shown in Figure 5.15. For reference purposes this version of the cull will be labelled the *vertex cull*.

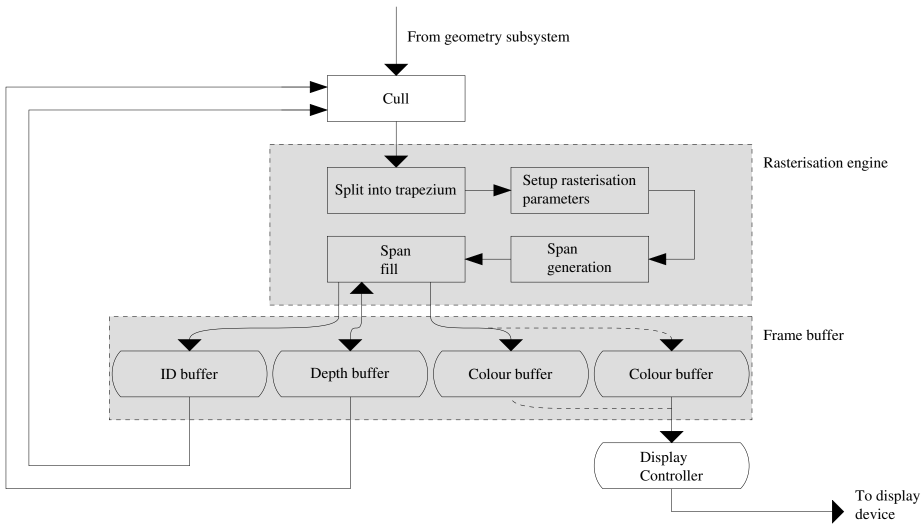

Figure 5.15: Rendering pipeline with cull stage

One way of removing the contention on the Z buffer (which already has to be both read and written by the rasterisation engine) is to maintain a table in the culling engine containing the furthest Z value of each polygon plotted. Instead of comparing the individual vertex Z values with the value present in the Z buffer, the nearest Z value of the new polygon is compared with the furthest Z value of the old polygon. This will mean that some polygons would not be culled where they would have been previously - especially if the occluding polygon is sloping steeply away from the viewer (Figure 5.16). For later reference this version of the cull will be called the *table cull*.

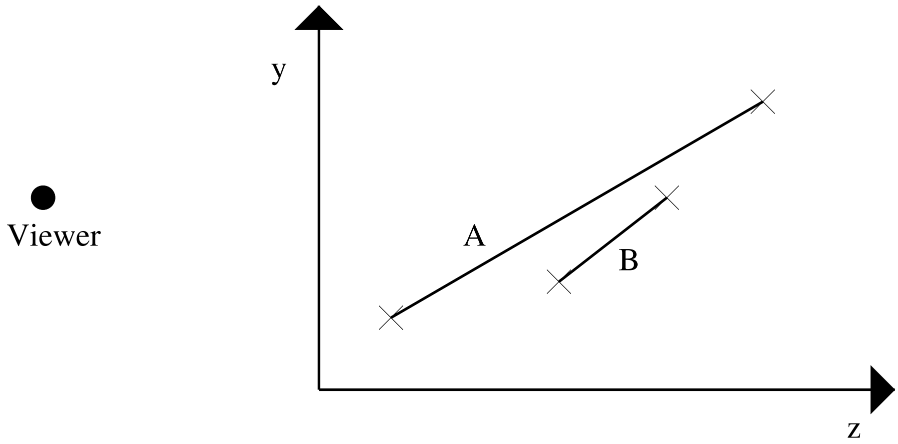

Figure 5.16: Loss of culling due to highly sloping occluder

Deciding which scheme is most appropriate for a particular rendering system design, involves the same variables which are present in the choice of any culling algorithm. The implementation is a trade-off of complexity (and therefore cost), potential benefit, and any degradation in performance if the culling algorithm is not effective.

5.6.2 Allocation of polygon IDs

It may be of concern to the reader that the depth of the polygon ID buffer (and hence the size of Z table in the table cull algorithm) places a hard limit on the number of polygons which can be rendered using this algorithm. Such a limit would be undesirable since it is always necessary to produce a correct and complete image. No such limit is produced since all that is necessary is to ensure that any polygons which can not be allocated IDs do not modify the polygon ID buffer while being rendered. If this is the case it is important to ensure that major occluders are rendered early in the rendering process and get assigned polygon IDs. There are various ways of increasing the effective number of polygon IDs. The simplest method is to return a flag from the rasterisation engine stating whether any pixels for a particular polygon passed the Z buffer test. If all pixels were hidden then that ID can be reused.

5.6.3 Simulation results

This section presents the results of simulations of the culling algorithm on the models described earlier in this chapter. Table 5.2 and table 5.3 show the results of both of the culling methods on the models. In these simulations the triangles were rendered in the order which they were presented to the system by the application. Most scenes show a small amount of culling has been achieved. In all cases the vertex cull performs better than the table cull since the later often cannot cull objects near a ‘sloping’ wall. Very little culling is achieved in either the Division bedroom scene or the radiosity scene. The reason for this in the Division bedroom scene is that there are very few objects in the scene to be hidden. Reasons for the culling algorithm’s poor performance in the radiosity scene are described in section **5.6.5.2**. It is interesting to note that the percentage of pixel writes saved is very small in all cases. It is expected that this is due to two reasons:

- Most primitives being culled are far from the user (in a different room, or away from the section the user is looking at directly). Perspective causes these primitives to be smaller.
- These culling algorithms require that a primitive is entirely occluded by *one* other primitive. Many large primitives which are hidden are often occluded by a number of separate primitives, while it is likely that only small primitives will be occluded by a single primitive.

	CADCentre test model	CADCentre plant	Division house	Division bedroom	Jack	Radiosity scene
% triangles culled	4.56	4.83	5.50	0.40	6.93	0.74
% pixel writes saved	1.44	0.80	0.26	0.00	0.16	0.26

Table 5.2: Results of table cull on original data

Table 5.4 and table 5.5 show the results of running the same simulations on lists of primitives sorted in front-to-back order. This increases the likelihood of occluding surfaces

	CADCentre test model	CADCentre plant	Division house	Division bedroom	Jack	Radiosity scene
% triangles culled	4.98	5.93	5.95	0.40	7.85	0.85
% pixel writes saved	1.46	0.93	0.26	0.00	0.20	0.26

Table 5.3: Results of vertex cull on original data

being drawn before the primitives they hide. The culling algorithm now produces a useful amount of culling. The algorithm is extremely effective on the CADCentre test model due to the low wall which occludes much of the detail in the scene (for this reason this model is not representative). The culling algorithms perform particularly well on the CADCentre plant model; Figure 5.17 shows a comparison of the depth complexity of

Figure 5.17: Depth complexity comparison using the vertex cull algorithm on sorted data

the CADCentre plant scene rendered without any culling and rendered with the sorted vertex cull. The figure was generated by recording the difference in depth complexity maps for a simulation without culling and a simulation with the sorted vertex cull. The resulting file was output in a grey scale image format (pgm) and was processed to enhance the contrast before printing. The darkest sections show where the most culling has occurred. This diagram shows that the culling algorithm has been highly successful in removing detail hidden in the office block.

The author was initially surprised at the increase in culling between the table and vertex cull on the sorted version of the Division bedroom data set; a depth complexity comparison revealed the vertex cull has managed to eliminate a number of primitives which appear to be part of the landscape outside the house.

These tables include two extra measurements - the mean depth complexity and triangle area after culling. The mean depth complexity is slightly reduced (when compared with the unculled or unsorted case) since less is drawn, although the effects of the culling are masked by large areas of background or walls which dominate the figures. In all cases the mean triangles area increases when compared to a rendering without culling. This is because most of the primitives that are culled are small. One anomaly is that on the Jack model the average triangle area is slightly smaller on the vertex cull technique than the table cull technique. This was not expected since the vertex cull technique generally culls a small amount more and is thus likely to cull more primitives, again most of which are likely to be small. By producing a depth complexity comparison of the model with table cull and vertex cull the reason was clear. The vertex cull had managed to remove some of the shaded tiles which make up the floor of the scene below the model. These tiles are fairly large and thus removing them reduces the mean triangle area.

	CADCentre test model	CADCentre plant	Division house	Division bedroom	Jack	Radiosity scene
% triangles culled	63.81	15.81	32.06	0.99	9.57	1.60
% pixel writes saved	16.03	4.54	2.05	0.00	0.44	0.35
mean depth complexity	1.113	2.922	1.634	2.648	2.340	1.553
mean triangle area	74.26	18.53	259.34	1451.17	49.68	89.20

Table 5.4: Results of table cull on sorted data

	CADCentre test model	CADCentre plant	Division house	Division bedroom	Jack	Radiosity scene
% triangles culled	64.08	18.60	33.12	3.36	10.19	2.39
% pixel writes saved	16.03	6.10	2.15	1.30	1.23	0.40
mean depth complexity	1.113	2.874	1.632	2.613	2.321	1.552
mean triangle area	74.80	18.87	263.20	1468.66	49.61	89.86

Table 5.5: Results of vertex cull on sorted data

5.6.4 Order of rendering

As with the hierarchical Z buffer technique described above, the order of rendering is very important. If major occluding objects are rendered too late very little will be culled. Obviously sorting the primitives before rendering is not practical, so a number of alternatives have been considered:

- Use with grid cell space subdivision

  One advantage of the grid cell technique is that using appropriate traversal algorithms it is possible to visit the cells in front-to-back or back-to-front order at no extra expense. This would help sort the primitives to gain the most advantage from this technique. The grid cell technique could be used to extract only objects within the viewing frustum and this clipping algorithm is used to remove hidden objects.

- Sorting at each level of the hierarchy

  It would be possible to modify the model hierarchy to include a bounding box for each sub-tree. It is then a simple matter to modify the model traversal algorithm so that the sub-tree whose bounding box is closest to the user is traversed first. Since there are typically only a small number of objects at each node of the hierarchy the sorting process should be fairly fast. It would also be possible to perform the sort only on the upper few levels of the hierarchy and plot the lower levels in the most convenient order; it is expected that the lowest levels of the hierarchy will contain fine detail whose ordering will be mostly insignificant.

- Frame-to-frame coherence

  As in the hierarchical Z buffer scheme it would be possible to use visibility information from the previous frame to aid in determining the best rendering order to use. A flag could be added to each hierarchy node which would be set if any object at that node (or its children) was visible at some stage in the rendering process (i.e. a pixel passed the Z test). Those nodes which do not have this flag set could be rendered last.

5.6.5 Effectiveness of this technique

This section looks at how effective the technique is in general and some ways of improving it.

5.6.5.1 Insignificant occluders

Small objects rendered in front of occluding bodies can cause the culling technique to be less effective than expected (Figure 5.18). If these primitives are rendered in the order C,A,B or A,C,B (but not A,B,C) the culling algorithm can not cull B since the information in the polygon ID buffer states that one vertex is obscured by A and the other is obscured by C. Although no simulation figures have been generated to investigate this problem it is expected that this case happens frequently. One example of such a situation might be in a house where objects in the current room are rendered first, then the wall at the back of the room, and then objects in the next room. An object in the present room could be an insignificant occluder which stops the algorithm culling out objects in the next room. A solution to this problem is not to allow insignificant primitives (such as C)

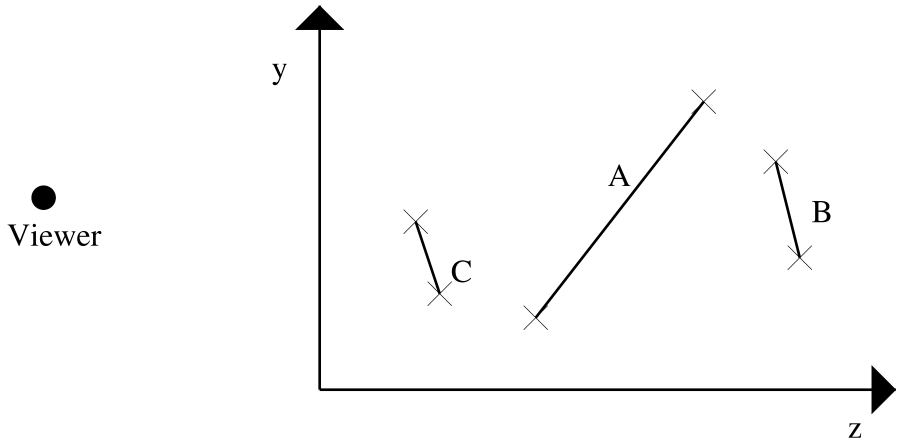

Figure 5.18: Culling lost due to an insignificant occluder - C

to modify the ID buffer. What constitutes an *insignificant primitive* could be determined by the primitive's area (or something simpler such as its x or y extent). Alternatively it is likely than an application has knowledge of which objects in the scene are likely to be major occluders (such as walls of rooms) and can thus influence the culling algorithm. Many current graphics libraries do not provide any ways for the application to provide *hints* to the rendering system - this issue is described in more detail in section **6.2.2**. Another method of handling this situation is to enhance the depth table held in the table cull algorithm to include information relating primitives such as A and C; this would however be somewhat difficult.

5.6.5.2 Multiple polygons

In the radiosity scene very little culling was performed. Although there are some large surfaces, the surfaces are subdivided into many small primitives to produce shading variation. Since the culling algorithm requires that *one* polygon acts as the occluder the subdivision effectively disables most of the culling. This problem also occurs with walls which are represented by a number of triangles. A solution to this problem would be to use one polygon ID for the entire surface which has been subdivided, since its occlusion characteristics are effectively equivalent to one polygon.

5.6.5.3 Tessellation

Tessellated objects (such as the cylinders in the CADCentre models) suffer from similar problems to the subdivided polygons of scenes lit using radiosity. In the case of the table cull method it would be possible to assign the same ID to the whole of the object which is being tessellated and then store the furthest Z value of the whole object in the table. A disadvantage of this technique is however that a large cylinder could not obscure objects inside itself, only behind it.

5.6.5.4 Culling the hierarchy

A problem with the algorithms described so far are that they operate at the level of the primitive; this is too costly and does not save enough when a primitive is culled. It is however possible to use this technique to perform culling of sections of the hierarchy using a technique similar to that used in the hierarchical Z buffer scheme; two separate methods have been considered; the first method is to implement the octree based scheme described in **[GrKM93]**. The time consuming part of the octree based algorithm described in that paper, is the time taken to test the octree cubes against the hierarchical Z buffer. Using the polygon ID technique, the test for each octree cube is a matter of testing the values in the polygon ID buffer and Z buffer at the position of the vertices of the cube projected onto the viewing plane. Unlike the technique described by Greene, Kass and Miller this technique has a much more predictable execution time and simple pixel plotting time is not significantly affected. The second method of larger scale culling is to test the bounding cubes of each sub-tree of the hierarchy before rendering the objects in that sub-tree, if the bounding cube is not visible it is not necessary to descend into that sub-tree. This technique is simple and fast and can cull large sections of the hierarchy. As with any culling algorithm there may be pathological cases where very little is culled and the time taken by the culling algorithm may itself be significant. One possible way of avoiding this is to have a flag on each node of the hierarchy stating whether culling should be carried out on it. The flag could be controlled by a simple heuristic based on the previous frame. One such heuristic could be to turn on culling if many objects are hidden by depth buffering and to turn the clipping off if only a few primitives are culled.

The advantage of both these methods is that they operate higher up the rendering pipeline and thus each cull test has the potential to remove a larger section of the scene. In addition, since the culling algorithm is so high up the pipeline, very little work has been wasted on the object that is about to be culled.

5.7 Summary

The scenes used by many VR applications are extremely complex; and are likely to increase in complexity. Much of this complexity is not visible to the viewer, either because it is far off, because it is out of his view or because it is hidden by other objects. Level-of-detail switching and culling enable much of this complexity to be removed before the rendering stage, and are thus essential tools in using rendering hardware efficiently. It is however important that these culling and level-of-detail selection algorithms do not place an even greater load on the host. The culling algorithm presented in this chapter is simple, does not consume much processing time and is easy to implement in hardware. As with all other culling algorithms the author is aware of, the fraction of the scene culled by the algorithm is difficult to predict; this makes maintaining a constant frame rate difficult.