Chapter 3

Rendering Surface Models

3.1 Surface Rendering

We now have the preliminaries settled: how to build a model, and how to transform it. Next we have to discuss how to produce an actual picture. This process is called rendering. We have so far only considered surface models, so we are about to describe surface rendering. (Renderers are tailored to a particular kind of model representation.)

We start from a mesh model, generally formed from triangular facets. We shall look at producing a shaded colour picture, which will involve

- Pre-processing the model, to remove parts which cannot be seen in the final picture. This involves both clipping and back-face culling
- Lighting the model. This includes the use of light direction and intensity; but it also includes surface reflection calculations (different surfaces have different reflective properties).
- Projecting from 3D to 2D: projective transformations such as perspective.
- Hidden surface removal: only at this stage do we discover which objects obscure parts of other objects.

We will also look at how to incorporate texture detail on surfaces, for greater realism.

3.2 Pre-Processing the Model

3.2.1 Terminology

When we discuss the removal of triangles which we cannot see, we need to use four related but different concepts.

Culling triangles is the general term for getting rid of unwanted triangles.

Clipping means cutting a triangle, usually against a plane (3d) or a line (2d). We use this to remove the unseeable part, for example when a triangle is partly off-screen. It can be done in the 3D model or on the 2D screen.

Back-face removal is the removal from a 3D scene of those triangles which are facing away from the viewer. It is done for efficiency reasons: we don't want to process them any further because they cannot contribute to the final picture.

Hidden surface removal is the removal of any part of a surface which is occluded by another surface, from the viewpoint. This is the most complex to implement: in principle it requires every triangle to be compared with every other triangle. Contrast this with back-face removal, which can be done with knowledge of each triangle in isolation. HSR may require clipping, because part of a triangle may be visible. We will always do back-face removal first, because that dramatically reduces the scale of HSR calculation.

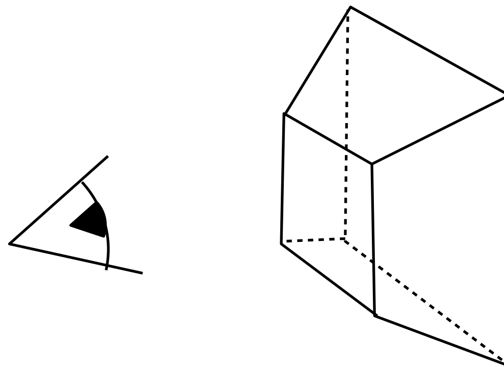
3.2.2 Culling and Clipping in 3D

We wish to cull those parts of the model that cannot be seen because they are outside the viewing volume.

Culling is very useful for large models: if we can discard (i.e., not bother to consider) half the model because we know it cannot be seen from this window, then we have halved the amount of computation necessary. Indeed culling can save a huge chunk of time because often *most* of the model is not viewable, for example in a VR environment.

Most facets will be entirely inside the viewing volume or entirely outside. We accept the former for further processing and reject the latter. The problem case is those polygons which are partly viewable. We have to scissor-away the section of each which cannot be seen: hence the term *clipping*. In 2D, clipping is simply a calculation of what intersects the edges of the display/picture.

It is not quite so simple in 3D:



The volume that the eye can see is a truncated pyramid. The near plane is the screen, the far plane at some convenient distance. There are thus six *clipping planes* which bound the *viewing volume*. The mathematical term for a solid sliced by two parallel planes is a *frustum*, so this is also called the *viewing frustum*.

Clipping is usually done by a fast subdivision method: for each edge in the model we check against a clipping plane and find whether the two ends are

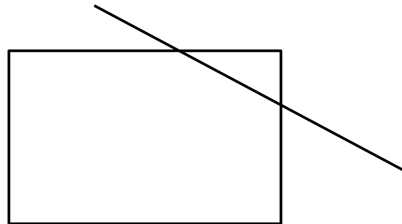
(a) both outside the viewing volume: throw it away

(b) both inside the viewing volume: (potentially) draw it

(c) one inside one outside: bisect the line and recurse on each half

Testing for inside or outside is a matter of plugging the coordinates into the equation for the clip plane, and seeing if the result is positive or negative.

We have to test against each clip plane individually and combine the results. It could be that the ends of an edge are outside the viewing volume, but the middle of the edge intersects



An alternative to bisection is to compute the intersection of the edge with the clip plane, and then each sub-edge will fall into category (a) or (b).

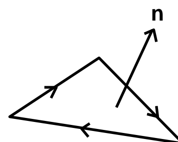
You can also see from this why we cannot rely exclusively on 2D clipping. A 2D clip is performed against the edges of the display i.e. after we have projected the 3D scene down to 2D. Triangles which are behind the viewer would project onto the screen as well, so we have to cull them at the 3D stage.

3.2.3 Back-face removal

In a similar spirit we can throw away any facets which are facing away from the viewer. This is know as *back-facing culling* or *back-face removal*.

The reason we can do this is that most real objects have a non-negligible thickness. For example, both sides of a wall have to be modelled to give an accurate shape to a building. However, if I am standing on one side of a wall, I cannot see the other side of it. So we can throw away the polygons which are facing away from the viewer, reducing the number of polygons to be considered when producing the picture. We do this to improve the efficiency of later stages.

We arrange the model so that every facet has an orientation, so we can determine what is its front, and what is its back. The simplest way to do this is to record the edges of the triangle consistently; for example, so that the vectors formed by each edge proceed clockwise around the triangle when seen from its front.



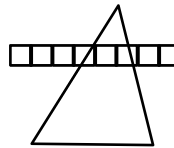
With this arrangement, the vector product of two successive edges will be a vector in the direction of the normal.

Back-face removal and 3d clipping are both pre-processing steps, needed before we can get down to the real business of rendering the model. They may be done together, in a single pass through the model.

3.3 Lighting the Model

We now look at the way we apply lighting (described in detail in the ray tracing chapter) to the patches that form the model. There are two separate issues which come together here. The first is the normal of the surface at the position being rendered (together with its colour and any other features that affect the way it reflects light). The second is the model we have for how the incoming light is scattered by a particular set of surface properties. It is the interaction of these two which determines the local effect.

First we assume that the renderer works on a horizontal scan line, producing the pixels for that line before moving on to the next line. At any point, therefore, the renderer only has access to a small slice of the model.

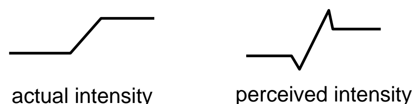


One thing to keep in mind: the mesh is a 3D structure, while the screen is 2D. In the above diagram, the triangle is in 3-space but the pixels are in 2-space.

3.3.1 Constant shading

This is the simplest method. We give each triangle a single intensity value. If we just use the underlying colour of the surface, there will be no interaction with the lighting and we will not be able to see the shape of the surface. Instead, we find the surface normal of the facet, use a diffuse lighting computation, and colour the whole triangle with the resulting colour. Each triangle has its own normal: those facing the light will be brightly lit, those facing more obliquely will be darker. In this way we will be able to see the shape of the surface.

This shading method is computationally cheap, and is good for a quick look at a model. One problem is that the individual triangles of the mesh are all visible. There is something worse: the *Mach band effect* whereby the eye perceives sharp intensity changes as dark and bright bands. This makes the edges look much worse.



This shading does at least reveal the real structure of the model: it really does consist of triangles.

Method Calculate the surface normal, then the cosine of the angle between it and the lighting direction (from the dot product of the directions: we don't ever need to know the actual angle itself). Shade proportional to this.

Vector Calculations

In the above, we made use of the vector cross product and the scalar (dot) product. A little bit of revision is helpful here. Suppose we know the coordinates of the vertices of a triangle. Let's also assume the triangle is not degenerate: the vertices are all different and are not co-linear. The vector defining the edge between vertex 1 and vertex 2 can be thought of as simply the coordinates of vertex 2 expressed with vertex 1 as the origin: $(x_2 - x_1, y_2 - y_1, z_2 - z_1)$. If we choose these vectors consistently around the triangle (from vertex 1 to vertex 2 to vertex 3 to vertex 1), they will go around clockwise as seen from one side.

Now it follows that any two edges of the triangle define the plane containing the triangle, useful in itself. The normal to this plane/triangle is, by definition a vector at right-angles to it and thus at right-angles to any pair of the edge vectors. The vector product of two vectors **a** and **b** produces such a vector and is given by:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k}$$

(Notice the way the subscripts are cyclic. The first term is in the x direction and uses y and z subscripts etc)

This also shows why the alternative name cross-product is sometimes used: calculating the x component requires terms in y and z. (Another way of formulating this is as a determinant, with the **i j k** in the top row, the a_x a_y a_z in the middle row and the b_x b_y b_z in the bottom row.)

Example The vector product of (2,3,1) and (-1,0,5) is:

$$(3.(5) - 1.(0))\mathbf{i} + (1.(-1) - 2.(5))\mathbf{j} + (2.(0) - 3.(-1))\mathbf{k}$$

which simplifies to:

$$15\mathbf{i} - 11\mathbf{j} + 3\mathbf{k}$$

In many applications, such a vector will need to be *normalised*; that is, we work out its length (the square root of the sum of the squares of the coefficients) and divide this into each component to give a vector of unit length.

We also made use of the scalar product, also known as the dot product or inner product. Using the same two vectors:

$$\mathbf{a} \circ \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

In other words, we just multiply the corresponding components and add up the result: there are no ‘cross’ terms.

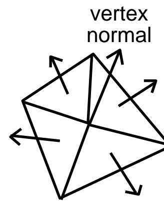
Example The scalar product of (2,3,1) and (-1,0,5) is 3.

You may also know that the scalar product of two vectors in a plane is equal to $a.b.cos(\theta)$, where θ is the angle between the two vectors and a and b are their lengths. This is useful when we need to know that cosine: it is equal to the scalar product divided by the product of a and b. Put another way, if we know the *unit* vectors in the directions of **a** and **b**, we can just use the scalar product to multiply them and the result will be the value of the desired cosine.

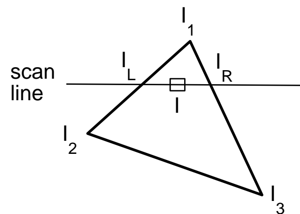
3.3.2 Intensity interpolation

Also known as *Gouraud shading*. This avoids the intensity discontinuity on edges (but the intensity derivative is not continuous). It gives the impression of curvature, and is widely used to help display curved objects that are really composed of flat facets. However it makes no real attempt to model the curved surface to which the mesh is an approximation.

Method Calculate the surface normal for each facet, then calculate vertex normals by averaging the surface normals for each facet meeting the vertex.



Now calculate intensities for the vertices with your chosen shading model. For each intersection of the triangle edge with the scan line linearly interpolate the intensity between the vertices at the end of the edge. Interpolate across the scan line between these edge intensities.



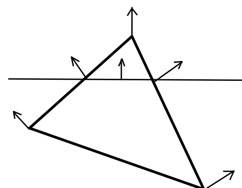
Thus, if we have computed intensities for the three vertices of this triangle as I_1, I_2 and I_3, we can compute I_L as the interpolation between I_1 and I_2, while I_R is interpolated between I_1 and I_3.

Then the intensity I for a pixel is interpolated between I_L and I_R.

- Gouraud smoothes across edges, which is generally what you want. If however you want an edge to be visible (for example, the edge of a table), you need to mark this in the data structure, and you can calculate two vertex normals, one for each side of the edge.
- The shading is continuous, but the derivative is not continuous. This can sometimes be seen as Mach bands, particularly during motion.

3.3.3 Normal-vector interpolation

This is the basis of *Phong shading*. This is similar to Gouraud, but instead of computing and interpolating intensities, you interpolate the normal vectors, and compute the intensity for each pixel. In this way, the interpolated vector approximates the curvature of the surface within each triangle. Shading is therefore calculated at every pixel.



This is particularly good in conjunction with specular reflection (highlights), models with highly curved surfaces (many small patches), and it gives quite good moving pictures. Mach banding is much reduced, but the cost is higher.

These days you can get hardware support for Gouraud shading, running at hundreds of thousands or millions of fully shaded triangles per second.

Texture maps (described in the ray-tracing chapter) are increasingly supported in hardware, too. This is why fast moving action games use surface rendering in preference to ray tracing, even though the latter is more realistic: you can get speedups of hundreds times by using purpose-built hardware.

3.4 The Projection Calculation

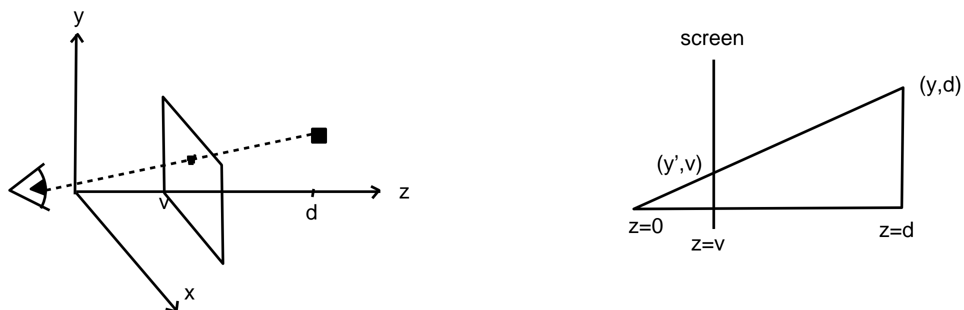
Here we describe how the 3D model is reduced to a 2D picture on the screen. Later we give a proper mathematical treatment of projection, but for the moment we give a simple overview.

There are many ways of projecting into 2D, we shall look at the two most popular.

3.4.1 Perspective projection

This is the kind of projection that we get from a conventional camera (with a normal lens). Objects further away from the viewpoint are smaller as we might expect, but such a projection does not leave angles or parallel lines invariant. That is lines that are parallel in reality may appear to converge in the projected picture.

This gives a good sense of depth to the picture, and is close to our everyday experience.



We want to find (x', y'), the screen coordinates of an object at (x, y, d) in the model. The derivation of x' and y' is identical, so we need only consider one, y.

We have a viewer at the origin, a screen at $z = v$, and a point in the model at (y, d). We want to find y'. By similar triangles,

$$\frac{y'}{v} = \frac{y}{d},$$

or

$$y' = \frac{yv}{d} = \frac{y}{d/v}.$$

Thus we are dividing by depth and v acts as a scale factor. So objects at a greater distance (larger d) appear smaller, and the rate at which objects diminish in size with distance is governed by the viewing distance v.

If v is small, the eye is close to the screen and we have a wide angle of view (like a wide-angle camera lens), the perspective is acute, so close objects appear unnaturally large by comparison with further objects. This is also true of normal vision by the eye, except the brain usually convinces us that things look normal. When we look at a photograph taken with an extreme wide angle it would appear correct if we could view it from very close up; we can't, because we cannot focus that close. So we have to view it from too great a distance, when the brain cannot fully cope and we perceive it as distortion.

The matrix for the perspective transformation:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/v \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Thus a point (x, y, z) becomes

$$(x\ y\ z\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/v \\ 0 & 0 & 0 & 0 \end{pmatrix} = (x\ y\ z\ z/v),$$

which corresponds to the real point $(xv/z, yv/z, v, 1)$ when we normalise. i.e. we have achieved divide-by-depth (z). We can also clearly see the scaling effect of v.

Note that

- the object is flattened to 2D, and lives on the $z = v$ plane, with scale factor v.
- we must clip (discard) points which are very close to the viewer, else we risk getting overflow when dividing by small numbers
- we must clip points with $z < v$ as these will also project on to the screen, including objects behind the viewer!

Lenses and Perspective

As an aside, if you are familiar with the effects of different camera lenses, you may find it helpful to see the analogy. What some computer graphics books call the viewpoint (as in the above description) is really the optical centre of the notional lens producing the picture. "Centre of projection" is a more accurate term than viewpoint. Computer graphics people project the picture onto a plane in front of the notional lens instead of onto a film at the same distance behind the lens but this is purely a convenience to get an upright image: it does not change the core mathematics.

The 'distortion' produced by a (good quality) wide-angle lens is not distortion at all but is a result of not viewing the photograph as close as the lens requires. Strictly, every photograph only looks correct from a particular viewing distance determined by the lens which took it. However, for very wide-angle lenses this means that the viewer would have to be so close to the photograph that they cannot even focus on it. Most of us just view it at a normal distance and think the picture looks distorted.

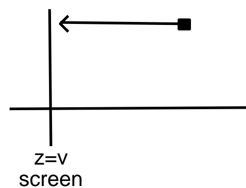
You may have seen a picture of someone apparently holding up the leaning tower of Pisa. Person and tower look about the same size. The photographer has chosen a viewpoint such that the perspective scaling of the two objects makes them look about the same size. The tower is much bigger but is also much further away.

The 'jumbo jet over a house near Heathrow' picture, which appears in many newspapers when noise, extended landing hours or a new terminal become an issue, typically shows a very large aeroplane apparently skimming house rooftops. Such pictures can give a false impression, when produced by using a long focal length lens. What such a lens does is allow the photographer (and the centre of projection) to be a long way away, while still getting a photograph which fills the frame. (Since v is an overall scale factor, we can zoom-in – magnify – by selecting a large v, thereby filling the frame even though we are further away.) Now the *relative* separation of the house and the aircraft, from the photographer's position, is much smaller than if the viewer

is just outside the house. In other words, the 'depth' value in the 'divide by depth' formula is closer to being the same for the two objects. In consequence, they appear at closer to their correct relative proportions (the jet really is much bigger than the house). However, such a photograph does not show what you would see if you were standing in the street near the house: the aeroplane would look much smaller. Even so, there are indeed residential streets under the Heathrow flight-path where no such trickery is needed: the aircraft really are just about to land and so are very close to the ground.

3.4.2 Parallel projection

This kind of projection is popular with engineers and architects: it preserves angles and parallel lines, and so is good for when you want to make things. It is a limiting form of perspective projection as we take v to infinity.



This is a very easy transformation, we simply drop the z coordinate and replace it with the position of the viewplane.

$$\begin{aligned} x' &= x \\ y' &= y \\ z' &= v \end{aligned}$$

We can still get foreshortening, but this is not a function of distance. Objects remain the same size on the screen regardless of their distance in the model.

The matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & v & 1 \end{pmatrix}.$$

3.5 Hidden Surface Calculation

In a 3D model, some objects may be behind others and thus will be partly or wholly obscured from a given viewpoint. When producing a realistic picture we need to determine those parts of the model that can't be seen, and not draw them.

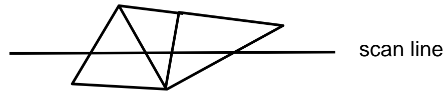
There are many ways to calculate what can be seen, but they really all reduce to a problem of sorting.

Our assumption here is that our model contains no self-intersecting surfaces: if there are, we can break it down into smaller triangles that don't intersect.

3.5.1 Scanline Algorithm

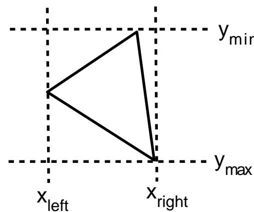
This is due to Romney, Watkins and Evans (1969), and displays a model composed of polygonal (triangular) facets on a raster display. Its strength is that it works with *spans* of pixels – a horizontal run of pixels from the same geometric object – rather than a pixel at a time.

Consider a faceted model, and think of the way a scan line traces across it.

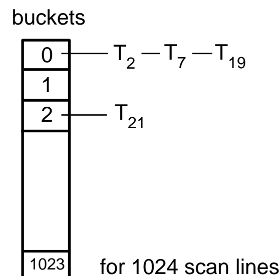


Notice that the picture remains the same (in terms of the visibility of facets) except possibly when the scanline crosses an edge. If this edge is where two different meshes overlap on screen, then it means that one object is obscuring another. If the surface to the right of such an edge is nearer than the surface to the left, then the visibility changes. Between the edges are spans of pixels where there is no geometric change, so the visibility cannot change. These observations lead to the following algorithm.

1 Remove all backwards facing facets (recall that facets have an orientation given by the direction of their edges). Such triangles cannot be seen from the viewpoint. Incidentally, we regard facets as having zero thickness, so to model concave objects like a cup we have meshes for both the inner and outer surfaces.



2 Now compute the vertical and horizontal extents, as shown above. Sort into *buckets* with all those triangles for which $y_{\text{min}} <= 0$, $y_{\text{min}} = 1$, $y_{\text{min}} = 2$, etc. Within a bucket list, sort the triangles left to right using x_{left}.



3 Create an *active list* initialised to bucket list zero. The active list will be the list of triangles that intersect the current scan line. We initialise this with scan line 0.

4 Draw the current scan line (see below).

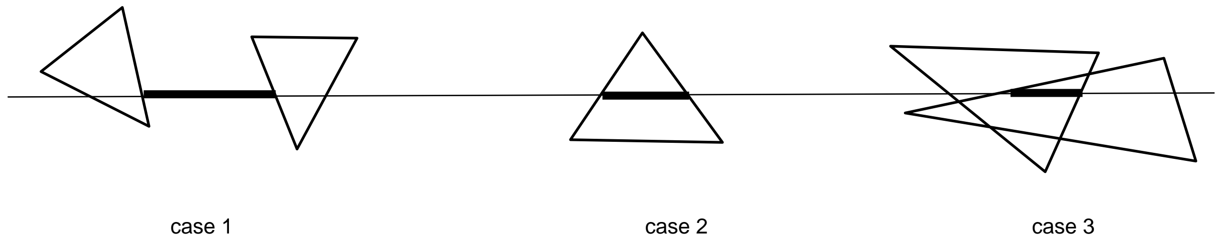
5 Move to the next scan line: update the active list by

(a) removing all polygons for which the current y now exceeds their y_{max}

(b) merging in x_{left} order those triangles in the next bucket (because their y_{min} says they are now on the current scan line).

6 Repeat until all scan lines done.

We just need to show how to draw a scan line, without drawing the hidden parts of triangles. We work along the active list, considering the intercepts of the scan line with the edges of the triangles. This produces a list of spans over each of which the visible surface does not change. There are three cases.



1. The span lies outside all triangles. Output a span of the background colour.

2. The span lies within exactly one triangle. Output the colour of the triangle.

3. The span lies within two or more triangles. For each of these triangles compute its z depth at $(x_{\text{left}}, y_{\text{scan}})$, and output the colour of the closest triangle. At this point we rely on the non-intersection of triangles.

Notes

- The computation of the z depths can be done incrementally across scan lines.

- Case 3 is the most costly because it requires a 3D calculation (remarkably, everything else is 1D) so we try to avoid the computation it incurs. Thus, if the active list remains unchanged across scan lines we know that the hidden surfaces cannot have changed. Further, there are usually only a few changes in the active list as we proceed, so large parts of it can be dealt with as in the proceeding case. Re-sorting intercepts into left-right order is thus best done with bubble sort (its one virtue is that it is fast with nearly-ordered data).

- If we are using constant shading, the natural output of this algorithm is a (colour, length of span) pair, which is a compact representation of a picture (run length encoding).

- The method is well-matched to Gouraud shading because we can incrementally calculate the left-right intensities at the same time as the left-right intercepts. The, instead of outputting a span of constant colour, we output pixels with the colour interpolated along the scanline. We can similarly applied Phong shading, by interpolating normals.

- Historical Note: The computation of the intercepts can be done incrementally down the screen. Integer addition was once *much* faster than floating point, so this gave a significant performance gain. This approach was used by Silicon Graphics in their 1993 “Reality Engine”. By 1997, this had been replaced by the “Infinite Reality Engine”, which instead used linear equations to represent edges. For more on the history of graphics hardware, see

  `http://www.cs.bath.ac.uk/~pjw/media/archive.htm`

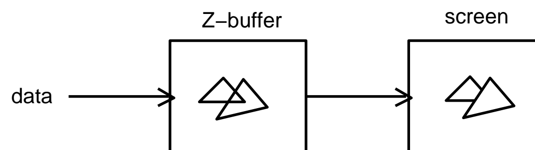
3.5.2 The Z-buffer method

As with many graphics problems, there is now a hardware solution to hidden surface removal. This version is the one implemented in graphics workstations and on PC graphics cards. In short, it is what you see in any full 3D game.

This involves extra hardware, the *z-buffer*. This is an area of memory with a location for each pixel on the screen. In this memory we store the z depth values of each pixel as we draw it. When an item is being rendered, we check each pixel it covers in the z-buffer: if the depth of the new item is greater than the value stored, we do not draw it to the screen (because the surface is obscured by something closer). If the item is closer, we draw it and store its z value in the z-buffer.

We initialise the z-buffer to contain ∞ (in practice, the largest distance the z-buffer can hold) for every pixel. Thus the final picture that appears on the screen shows exactly those pixels from the surfaces which are closest to the viewer. The simplicity of this approach (each item can be rendered independently with no elaborate data structures) coupled with its reliance on readily available hardware – mainly fast memory – has made it the dominant hardware solution.

It is not without faults: the depth of the Z-buffer determines the resolution in depth that it can distinguish, so items can be rendered in the wrong order; or may jitter in and out of view in a moving scene. Games designers minimise the problems by setting the depth range to suit the scene and avoiding closely spaced objects in scenes with long range visibility. Even so, you will see these faults from time to time. Also, the method is geared towards screen-resolution. This is usually what we want but it is a poor performer in the software implementations needed at very high resolution: the amount of Z-buffer goes up as the square of the image resolution. Further, each pixel is visited multiple times, once for every object which covers it, a square-law increase in processing time with the number of objects.



Shadows do not automatically appear when using surface lighting computations: either something is lit or it is dark. We can adapt the hidden surface algorithm to determine what can be seen from the point of view of the light source, i.e., what is illuminated and what is not.

3.6 Textures

So far we have concentrated on the basics of rendering a surface. Real surfaces are seldom visually smooth, they usually have some kind of visual texture to them, such as the pattern on a carpet, the woodgrain on a piece of furniture or just the scuff marks on a door.

Texture-mapping is extensively used in games. It can turn simple geometry into something apparently much more detailed. The inside of a room is just the interior of a box. The choice of texture can make it look like your living room or a cathedral. Textures can by changed dynamically, to give the effect of moving water. They can simulate a stream of light entering a dark room or produce a misty effect. They can simulate a wood or stone surface. They can add real detail from photographs or simulated detail pre-rendered from a complex computer model. If the 3D geometry is not important, they are the single most effective way of bringing visual complexity to a scene. You will begin to understand that it is very important that graphics cards can render textures rapidly.

There is a basic problem. Texture pixels (*texels* is the jargon word) are placed over a 3D surface; they do not in general cover exactly one 2D screen pixel. There may be many texels covered by each screen pixel (when the object is some distance away); there may be many screen pixels covering just one textured pixel (if we get very close to the texture); and of course the texture may not be aligned with the screen. Indeed, in general it will not even be in a plane parallel to the screen or be parallel to the pixel edges. Because of perspective, the projected texel will vary in scale even within a single textured area.

We therefore have to *resample* the texture to be sure that we get the correct visual effect. Imagine a tiled floor disappearing into the distance. In the foreground every tile is visible but in the distance the tiles are too small to be distinguished and we just see a blur. This is the effect we need to achieve. (See my picture collection for examples of this.)

In fact a black and white tiled floor (like an extended chessboard) is a good example because it allows us to think about the process of *sampling*. If we sample the tiled floor texture just by tracing a line through the centre of each pixel, to see where it hits the floor, then the foreground will be OK: we get lots of pixels of the same colour in succession, just as we expect. However, as we go into the distance the tiles appear smaller and the number of successive pixels of the same colour also gets smaller. There comes a point where, because the tiles are now smaller than the pixels, we might get several adjacent pixels appearing the same colour (black say), not because they belong to the same tile but because the in-between white tiles are never hit by the lines we traced.

At that point, there will be *bigger* areas of black than there should be; elsewhere nearby there will be bigger areas of white. The result is that, where we should have a very fine tiling in the distance, we will see a very coarse one, not what we expect. This is the *aliasing* problem, very rapidly varying brightness appearing “in disguise”, as slowly varying brightness. These sampling artefacts are very distracting, the more so in a moving scene where a slight change of viewpoint can produce a big change in the artefacts, giving an all-too-visible shimmering effect.

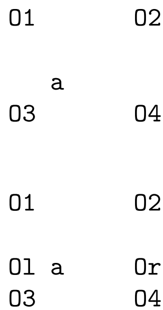
This arises because the process I have just described is one of *point sampling*. What we should ideally be doing is *area sampling*. That is, instead of tracing a line through the centre of the pixel, we should instead average everything that can be seen through a pixel-size square centred on that pixel. If we do that, then our fine tiles average out to grey, which is correct for a tiled floor so distant we can no longer distinguish the individual tiles.

This leaves one big problem to overcome: how can we do this in real time? In particular, pixels covering distant parts of the scene cover a lot more texture than pixels covering the foreground.

In fact, we cannot do that. Instead we can use a pre-computation technique (i.e. we can compute part of the effect when we build the model, greatly reducing what we have to do when we render the model). For example, we know in advance what any given texture will look like in the distance: it will be an average of what it looks like nearby.

To get the idea of how we do it, suppose we are only interested in representing textures in a range between their natural size (one texel = one pixel) and half their natural size (the texture is just far enough away that a square of four texels = one pixel). Suppose we start with the natural size texture and produce a second texture from it by averaging four texels to make one new texel. This is something we can pre-compute. When we come to render the scene, we can cover with texture any object which is at either of the two extremes of the distance range we are considering. Near ones get the high-resolution texture; distant ones get the half-resolution texture. What do we do for objects in between? The obvious answer is some kind of average of the two extreme textures but how to do this?

We first note that our texels are sampled on a discrete grid. In general (despite my simplified example above), the screen pixels will not exactly align with these. So we first need to ask, even for the high resolution texture alone, how we create a pixel value which corresponds to the texture at *any* point in the plane of texels (i.e. not just the convenient grid points of the known



texel values). What we do in practice is a weighted average of the four texel values surrounding the sample point.

Suppose the diagram shows the texel values at the 'O' positions and we want to know that value at the 'a' position. (I am assuming that the display and texture scanlines are horizontal on this diagram but the principle extends to any angle of texture). We first perform weighted averages vertically (i.e. at right angles to the scanlines), to find texture values Ol and Or (left and right):

The weighting for Ol uses the distances from O1 and O3 to Ol: the nearer O3, the greater the proportion of the value at O3 that we use; the nearer O2, the greater the proportion of O2. We don't use the values O2 and O4. Similarly we get Or from only O2 and O4, weighted by distances from them. (This is the same bi-linear weighted average technique we used for Gouraud shading.)

Then we perform a distance-weighted average horizontally (i.e. along the scanline)of Ol and Or, to get the value 'a'.

What have we achieved so far? We know that if we trace a line through a given pixel on the screen, then we can say what the texture value ought to be, if the surface it hits is at the (near) distance where the high resolution texture is defined.

If on the other hand, it hits a surface at the distance where the half-resolution texture is defined, then we could perform the same calculation but on the half-resolution texture map. We will still be averaging between four values. Call this resulting value b. Now, if the surface is somewhere between these two *we perform a distance-weighted average between the two texture values a and b.*

This technique is known as *mip mapping.* It readily extends to a succession of texture maps, each half the resolution of the previous one, in order to cover a wide range of depth values. All texture maps can be pre-computed, only the weighted averages have to be computed at render time. The computation requires bilinear interpolation, so PC graphics cards provide direct support for bilinear interpolation, with on-card memory for the multi-resolution textures in the current scene.

(For the curious, "mip" is an abbreviation of the Latin tag, "multim in parvo", many things in one place. This is a reference to the earliest implementation in which various resolution textures, separated into R,G,B components, were packed into a single array. You don't need to know this!)

It is important to realise that we can use this not just for surfaces parallel to the screen but also for surfaces which extend in depth (such as our tiled floor). We just choose, at each pixel, the right pair of maps (according to the distance where the ray hits the textured surface) and appropriate weightings.

In fact we can do better still. What should really determine the pair of maps to use is not the distance but rather the size of the pixel when projected onto the surface. For example, consider

a cube with one face almost but not quite parallel to the screen. Nearly at right angles to the screen is another face. We can see all of it but it is greatly compressed because we are seeing it almost edge on. Both faces fall in the same distance range but if both are textured with the same map, then both will show just as much texture. We expect the texture in the scarcely visible face to be squashed over just a few pixels. In contrast, the nearly square-on face shows its texture over a much larger part of the screen. Clearly we need to choose different maps for these two faces yet both are at more or less at the same distance from us. There is nothing wrong with our earlier analysis, it is just that we were only thinking of a single plane, in which case the size of the projected pixel is indeed proportional to distance. We now understand it a little better.

3.7 Conclusion

Here is a summary of the process for surface models and scan line rendering.

We need a model; a viewpoint and viewing direction; lighting specifications. We process by

- viewing transformations
- breaking up intersecting surfaces
- back face removal
- hidden surface removal
- shading
- output to display

One more point: we have assumed in the above that the depth is measured along the z axis. Or course in general this is not true because the viewer is free to roam around the model space. So we will need to transform the model so that the viewpoint is at the origin and the viewing axis is along the z axis. This is a translation followed by a rotation.

Of course, much hardware in modern computers is dedicated to rendering surface models (e.g. hardware support for shading and texturing), so there are also techniques to produce surface meshes from other surface representations and also from solid models. Where rendering speed is the prime requirement, this is a good way to go and mip-mapped texture greatly extends the visual effect. If visual quality is the prime need, then specialist renderers geared directly to the model representation do a better job than the kind of renderer considered in this chapter. We will see an example of this shortly, where ray-tracers are used with solid models. However mesh rendering used in a different way has produced the highest realism to date, with the radiosity technique, and we will also discuss this.