MANCHESTER 1824
The University of Manchester

COMP37111: Advanced Computer Graphics

Volume Rendering

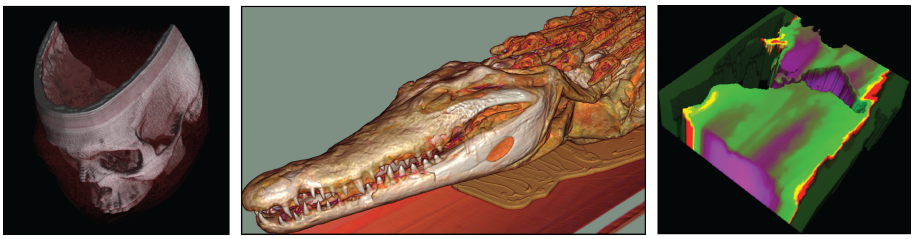
Steve Pettifer

November 2020

Department of Computer Science
The University of Manchester

1 Volume Rendering

Almost every form of rendering we've looked at so far in this course unit and its second-year predecessor has revolved around drawing an object's outer surface. This isn't surprising, since most things that we'd want to render can be represented nicely this way, whether that surface is generated parametrically or as a mesh of connected polygons. But a special class of scenes exist where we are less interested in the outer surfaces of things, and more concerned about what's 'inside'—and this is where Volume Rendering (M. Levoy, 1988) is useful.



(a) (b) (c)

Figure 1: Three examples of Volume Rendering from different kinds of data: (a) A volume rendered cadaver head using view-aligned texture mapping and diffuse reflection; (b) a mummified crocodile re-created from CT data and rendered using volume ray casting; and (c) properties of the worlds' oceans rendered from a simulation. Composite of images created by **Sjschen**[W], **stefanbanev**[W] and James Marsh.

Volume Rendering usually starts with a 3D data-set. This could be generated synthetically as the output of a simulation (say of fluid flow, or climate behaviour), but is perhaps more commonly acquired by scanning or measuring some real world phenomenon in a regular grid pattern. In the case of medical applications, the kind of data we're talking about here is generated from **CT**[W], **MRI**[W] or **MicroCT**[W] scanners; in a geographical context, the data could come from the use of **sonar**[W] or **ground penetrating radar**[W]. But anything that generates regular 3D grids of data can be used as a basis for volume rendered images. The important point here is that it doesn't really matter what properties the data set contains, what they represent in the real world, what units they are in, what the resolution of the samples is: as long they form a regular 3D array of data points, the techniques we'll explore here will be able to render them.

We're going to look at two different approaches creating images from volume sets, called 'Direct' and 'Indirect' Volume Rendering (it'll become obvious why they're called this later)—but both start with these 'volume sets', which are just 3D arrays of data. The 3D cells of such data sets are referred to as 'voxels' (the volumetric equivalent of pixels).

2 Direct Volume Rendering

Direct Volume Rendering creates images by casting rays into a volume set. Rather like Ray Tracing, we shoot these rays out from the eyepoint through a viewplane and into the scene (Figure 2).

Unlike Ray Tracing where we typically spawn new secondary rays as the primary rays intersect with object surfaces, in Volume Rendering we allow the original primary ray to penetrate through the scene 'into' the data, and we accumulate the effects of colour and transparency along the ray to work out what coloured pixel to plot on our viewplane as shown in Figure 3 (we'll look at an algorithm that combines the effects of all the opacity/colours visited in Section 2.2).

But where do we get colour and transparency values from if our volume set is made of

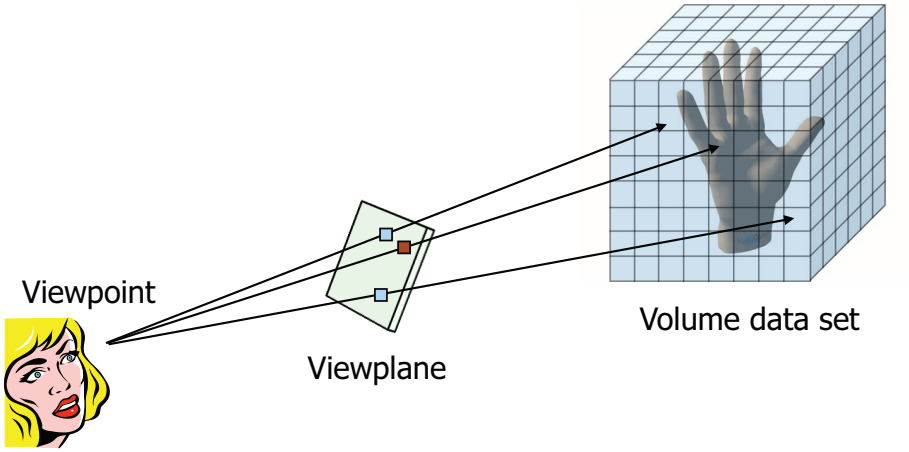


Figure 2: Simple Volume Rendering

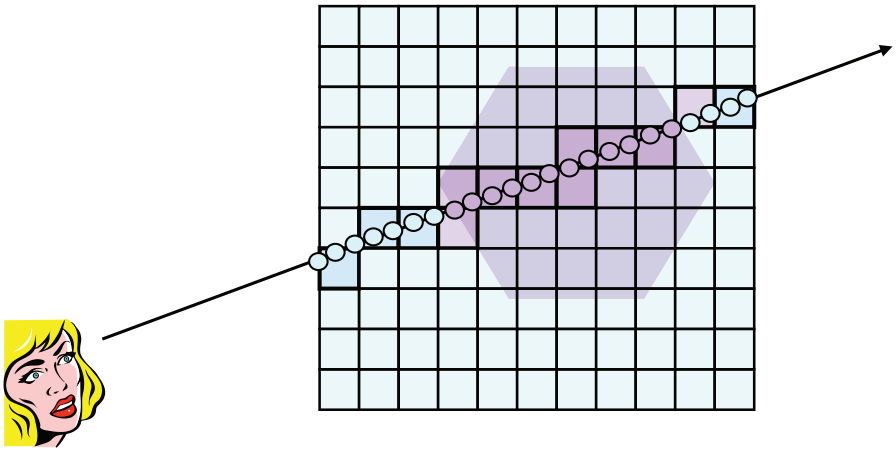


Figure 3: A ray cast from the eyepoint through a 2D representation of a voxel set; as the ray passes through the grid it accumulates colours from the different cells, depending on their opacity. If sampling along the ray is done once per visited cell, there is a danger of aliasing occurring, so it is common to sample along the ray at sub-voxel intervals.

arbitrary data? In the case of a **Functional Magnetic Resonance**[W] Scanner for example, our data points are some measure of of the strength of the magnetic signal from hydrogen nuclei in water at various points in the subject being scanned; in the case of sonar the values are representations of the time taken between sending out a sonar 'ping' and it being reflected back to the sensor. Whatever source they came from, they don't have any kind of colour or opacity associated with them, so we need to make that association ourselves. Generally speaking when using Volume Rendering, it's not to create photorealistic images anyway, but rather to see the 'inside' of something that you can't see under normal conditions, like the inside of a brain or a mountain, and its usually because you want to identify some special phenomenon that would otherwise be hidden (like a tumour in tissue or a fissure in rock). So making up colour schemes is usually fine.

The first step in generating colour and opacity values for our voxels is to classify the different 'materials' present in our data. The term 'materials' is used loosely here—it may of course be that there aren't any distinct materials involved if your data represents something continuous like the temperature of a liquid or something amorphous like water itself. But we'll use the example of an MR scan of a human body, so we'd expect to find resonance values that

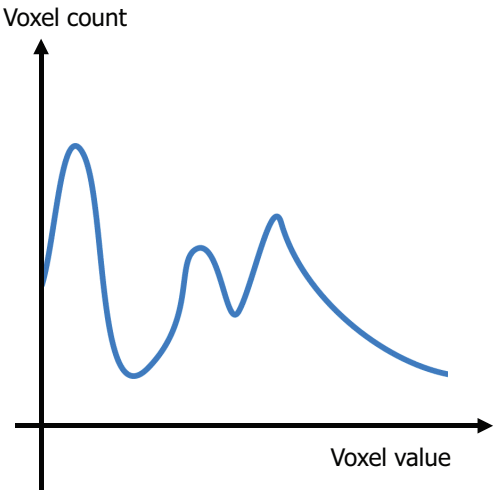


Figure 4: A plot of voxel value against frequency; peaks naturally occur around values that represent different 'materials' in the underlying data.

correspond to things such as bone, skin, muscle tissue, fat and air. The question is how to we determine which is which?

For now we'll assume that each voxel contains only one value. We begin by calculating a histogram of voxel frequency, i.e. for each data value, how many voxels have that value? We then produce a probability distribution to represent the likelihood that a given voxel value corresponds to a specific material or mixture of materials (imagine for a moment that the thing you've scanned consists of, say, just bone and air; we'd expect to see a lot of data points that have 'boney' values, and a lot that have 'airy' values, and some where intermediate values where we happen to have sampled a region of space that has a bit of both). A typical probability distribution function is shown in Figure 4. The peaks in this graph are likely to correspond to different types of materials commonly found in our data sets—the troughs between them are going to be where we've sampled transitions between one material and another (or could potentially be some other infrequently occurring material). For now let's assume that our scene consists only of air, fat, muscle and bone. By looking at the peaks in our graph, we could decide where to draw the distinction between one material type and another (and we'd count the tail of the graph as being anything that's not air, fat or muscle – in this made-up example that would be bone). Figure 5 shows a possible classification of the different voxel values.

For each material type, we now want to assign a different colour, as shown in Figure 6. Then we can set opacity values depending on what materials we want to be visible or invisible. For example, suppose we want to view only bone, without any of the clutter of muscle or fat. In this case we'd set the α-value of all voxels that have been classed as being bone to 1, and we'd want to make all other materials totally transparent (i.e. set their α-value to 0). If we want instead to see muscle displayed partially transparently and overlaid on top of the bone, we'd just set the corresponding α-values as shown in Figure 7.

2.1 Trilinear Interpolation

When calculating the value at a sample point along the ray, we could just use the value of the voxel we're passing through. Though this is quick, even with sub-voxel sampling distances along the ray it can still lead to aliasing effects, so it's usually better to interpolate values from neighbouring voxels in 3D to get a more representative sample value. This process is called **Trilinear interpolation**[W], and is shown in Figure 8. It sounds a bit grand, but the idea is quite

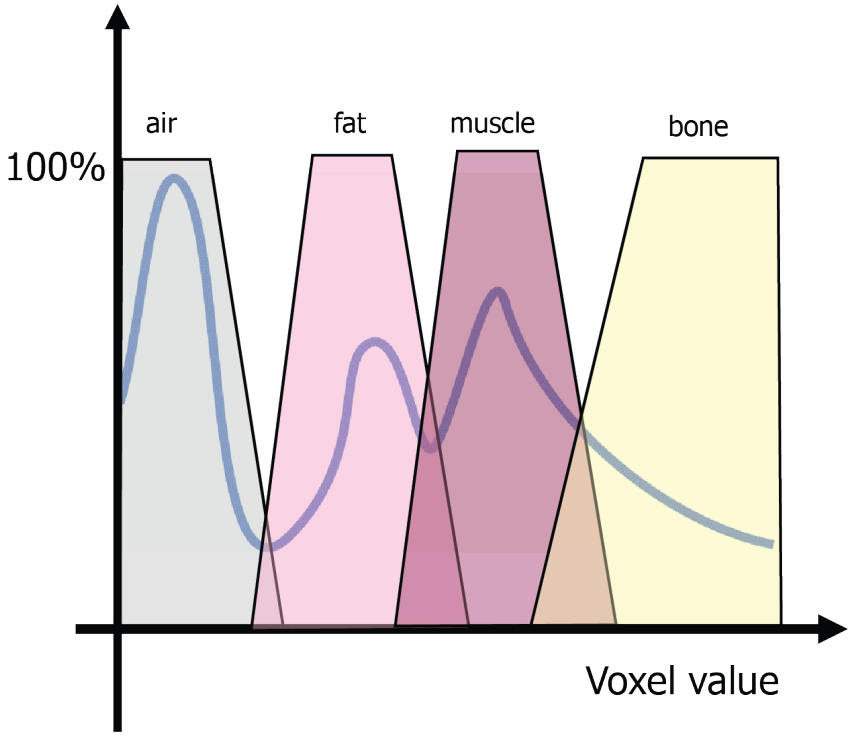


Figure 5: Voxels can be classified as representing different materials based on the peaks in the voxel/frequency graph.

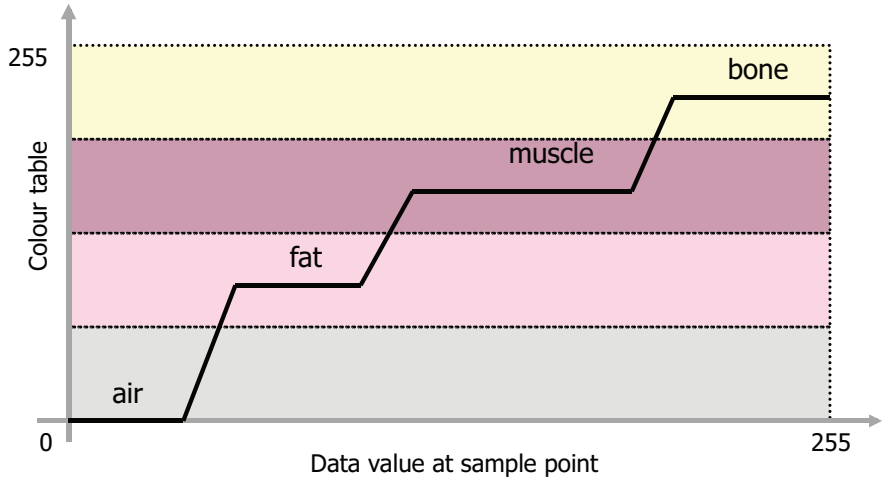


Figure 6: Once different voxel ranges have been associated with particular materials, the colour of those voxels can be set to represent the different materials.

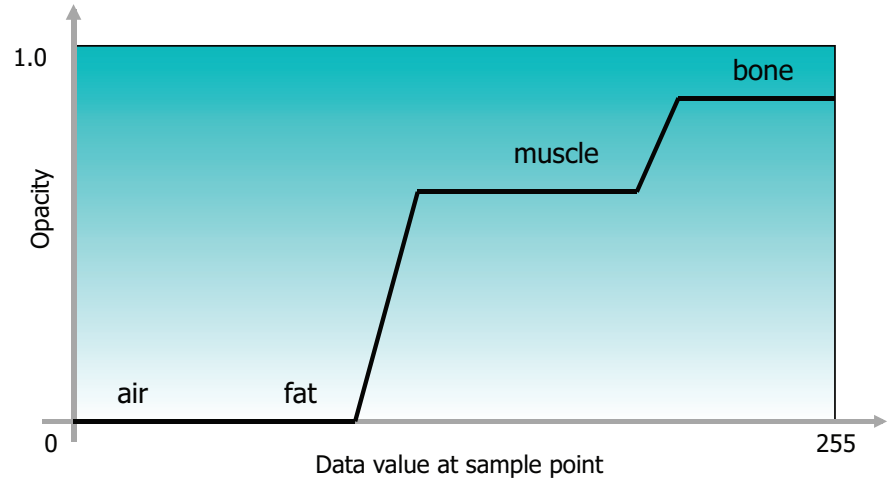


Figure 7: Once different voxel ranges have been associated with particular materials, the opacity of voxels in those distinct ranges can be set to give different rendering effects. In this example, all voxels in the ranges representing air and fat have been made transparent, muscle is set to partially transparent, and bone to nearly opaque.

simple: all you're doing here is 'averaging out' the values around the data point, biased in the three different directions.

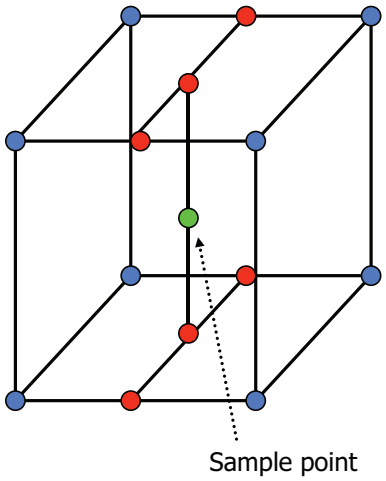


Figure 8: Trilinear interpolation

2.2 Computing the colour

With the colour and opacity values calculated, we can begin the process of casting rays into the voxel set and accumulating the pixel values on the viewplane by compositing along the ray. To calculate the effect a particular sample point will have on our final pixel's value, we need to work out the effect that of other values along the ray that leads to it (see Figure 9).

The general idea here is that you keep track of an accumulated colour, and an accumulated opacity value. Every time you step along the ray you take into account how opaque a voxel is, and its colour, and add this effect to the accumulated values until you reach a certain 'terminal opacity' (i.e. the ray has passed through so much material that it cannot penetrate any further).

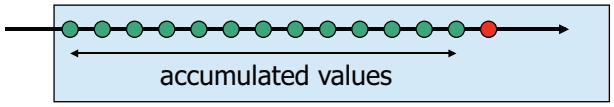


Figure 9

```
accum_colour= 0;
accum_opacity= 0;
/* find first sample point inside the volume */
while ((accum_opacity < terminal_opacity) &&
       (sample_point_inside_volume))
{
    /* eval colour and opacity at sample point */
    accum_colour+= (1-accum_opacity) *
    opacity * colour;
    accum_opacity+= opacity;
    /* step to next sample point */
}
```

Figure 10: Pseudo code for colour and opacity compositing along a ray.

This can be achieved by simple pseudo code shown in Figure 10.

If you render scenes using this approach with suitable colour and opacity values, the results appear rather 'flat', somewhat resembling X-ray photography (albeit, a strange kind of coloured X-ray, e.g. Figure 11). For some applications this is fine, but the absence of local surface shading loses some of the 3D nature of the scene. The problem of course is that we don't have any 'surfaces' in our data to shade: all we've got are the underlying data points, and some colour and opacity values that we've added ourselves.

In DVR, it's possible to create 'fake' surface normals from the data by interpreting changes in voxel values over the three different dimensions as giving the x, y and z components of a value that can be used as a kind of surface normal vector. Even though the vector's direction doesn't mean much in any physical or absolute sense, if the underlying data vary smoothly according to 'surfaces' that exist in the real world, then this vector will vary smoothly too, in some sense representing that surface. And that's all we care about in most cases (since inside a body or in the middle of a lump of rock, any notion of 'realistic lighting' is moot anyway!). By looking at these value gradients in three dimensions (Figure 12 shows this approach in 2D), we can create a faux surface normal vector that can be plugged into a Local Illumination calculation to give plausible surface shading, without actually having to pin down a definite surface.

2.3 Indirect Volume Rendering

In Direct Volume Rendering, any 'surfaces' that appear in the rendered image are implicit—they are effectively visual artefacts of the process and exist only in the mind of the viewer, rather than being something that's been calculated specifically or represented explicitly in the algorithm. As humans, we can see them, but the computer doesn't really know that its drawn surfaces. An alternative approach to creating images from volume data—called Indirect Volume Rendering—explicitly identifies surfaces in the volume set, converts these into polygonal meshes, and then uses 'traditional' polygonal rendering and local illumination techniques to draw these on screen.

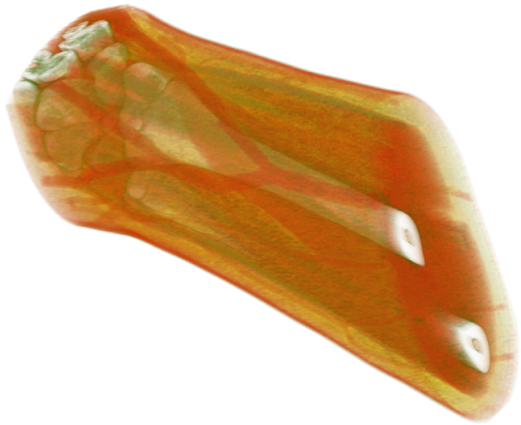


Figure 11: Rendering volume data without any kind of surface reconstruction leads to 'flat' images resembling X-ray photographs; without surfaces there are no specular highlights, for example, which would give visual cues as to the curvature of the object in 3d. Image by **Sjschen**[W].

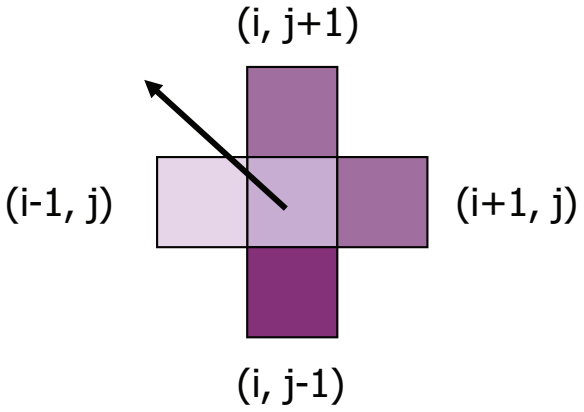


Figure 12

You're no doubt familiar with the use of **contour lines**[W] on maps to identify changes in height; for example in Figure 13, every time you cross one of the wiggly lines, you will have gone up 20 feet in height. Similar techniques can be used for maps to represent many other values; temperature, strength of geomagnetic fields and so forth, and the idea can be extended further to represent any of the kind of properties we'd be storing in a volume set: the important concept is that at any point along the line, the 'value' (whatever it represents is the same). So, these are called 'isolines' (from the Greek 'isos' meaning 'equal'). Figure 14 shows an isoline for the value 55, drawn through a 2D grid of some arbitrary data. Notice that in figure 14 the line has been drawn as a series of straight lines joining the specific point where 55 would appear. Remember that the data are usually going to be sampled from a continuous medium in the real world, so its likely that drawing a smooth curve that approximately follows the value through the grid would give better results.

The process of identifying an isoline on a 2D grid is fairly easy to understand (Figure 14 probably tells you all there is to know about it!). Applying the same idea to 3D data is conceptually straightforward; instead of finding an isoline, we now need to find an iso*surface*, i.e. the surface that represents a particular value in our 3D data. It turns out this is actually quite hard

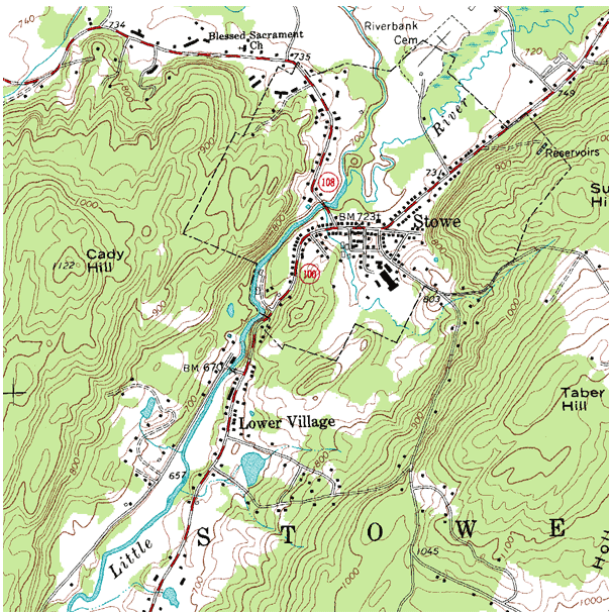


Figure 13: Topographic map of Stowe, Vermont. The brown contour lines represent the elevation. The contour interval is 20 feet.

to achieve in practice.

Look again at Figure 14. The 'algorithm' that was followed to generate the dotted isoline here is quite trivial: step through the vertical grid lines one by one, from left to right, and for each find the point at which the value '55' appears, and join all those points by a line. On each of the vertical gridlines, the isoline represents the point where the values change from being 'greater than the 55' to 'smaller than 55'.

Let's try to apply the same thinking in three dimensions: we're looking for a surface that lines in 3D space wherever the value 55 appears in our data; 'outside' that surface the values will be greater than 55, and 'inside' the surface they will be smaller than 55[1]. Figure 15 shows one possible configuration. The points on the right-most face of the cube we'll pretend are greater than our isovalue; so let's mark them as being 'outside' our surface. The points on the left-most face are smaller than our isovalue; so let's mark tose as being on the 'inside' of our surface. That means that for this particular set of 8 voxels, the change from 'insideness' to 'outsideness' happens on the four horizontal edges of the cube formed by our data. Exactly where on these lines the isovalue lies will vary, depending on the data points; but we know that for each line, the change takes place somewhere. We can therefore mark each of these points, and try to create a little patch of our surface. In this case the patch we'd need to create is easy; it's just a 4 sided polygon that joins the four data points.

Figure 15 is one of the simplest cases, however, where all the 'inside' values are on one side, and all the 'outside' values on the other. For this case it's obvious where the bit of surface should lie. But things needn't be as clean as this, and each of the 8 data points that form the cube's vertices can in reality be 'inside' or 'outside'. So…8 points, each of which can be in one of two different states gives us 2^8, or 256 different combinations of in and outsideness. In two of these 256 cases—where all the points are either inside or outside—the cube doesn't play any part in the surface since the isosurface doesn't pass through it. If we take into account symmetry, of the remaining 254 combinations there are 14 different distinct variants. These are

[1]the surface doesn't have to be an enclosed surface of course, so 'inside' and 'outside' aren't quite the right words, but they are less clumsy than 'to one side of' and to 'to the other side of'. So for now, if it helps, think of the surface we're talking about as forming a closed shape.

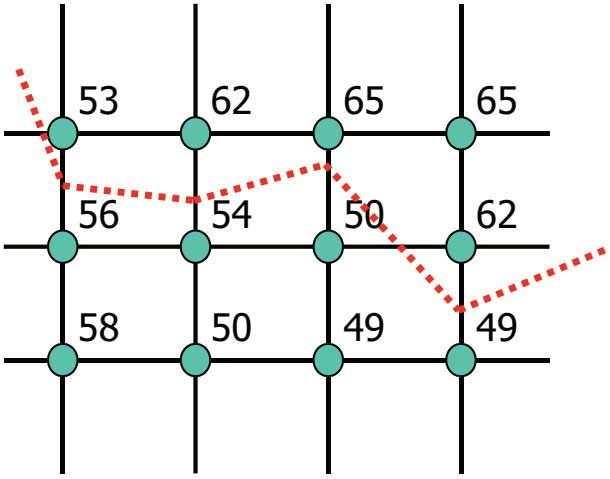


Figure 14: An isoline (red dotted line) drawn from left to right, linearly interpolating the point where the value 55 would be on each of the vertical lines (e.g. between 53 and 56 on the first vertical line, and between 54 and 62 on the second).

shown in Figure 16. In each case between 1 and 4 triangles gets added to the overall isosurface.

So to a first approximation we can imagine going through all the combinations of 8 data points that form the cubes we've been looking at, and accumulating a set of triangles to form the isosurface. We'd need to shade the triangles to give decent results, and this would mean finding surface normals for them. But which way does each triangle face? Although we've used the terms 'inside' and 'outside' for convenience, remember that this really just means 'to one side' or 'to the other' of our surface in some senses, and in any case whether we want the normals to face 'in' or 'out' would depend on whether we are looking at an artefact from the outside, or from within. And with Volume Rendering, both are possible and sensible! In any case, even though we could potentially calculate normals from the triangles we've generated, it almost certainly makes sense to calculate create these as 'faux normals' as described in Section 2.2 and then to linearly interpolate these to get normals at the triangle vertices.

But the problem is worse than this. Look at Figure 17, in which two cube vertices (marked with dots) are inside, and the others are outside our isosurface. Which of the two sets of triangles should we choose? Both are perfectly valid configurations that mark the two points as being on one side, and the remainder as appearing on the other. These ambiguous cases occur when adjacent vertices have different states, and diagonal vertices have the same state; and there are 5 other cases like this where we don't know whether to connect adjacent edges, or to connect accross to the other side of the cube. We can only resolve this ambiguity by looking at neighbouring cubes, and deciding which of the two options best forms a plausible isosurface. But what if one of the neighbours is also an ambiguous case? (see Figure 18 for an example of a mis-classification) The original algorithm for solving this problem was called 'Marching Cubes' (Lorensen and Cline, 1987) and though the details of the algorithm contained several errors and inconsistencies that have been corrected by later authors, the name stuck—informally at least—for the whole class of algorithms.

2.4 Proxy Geometry

An increasingly popular alternative to finding and rendering isosurfaces is to use 'proxy geometry' to recreate a volumetric image. This is a kind of half way house between rendering the image onto a viewplane (as with direct Volume Rendering), and creating a fully blown polygonal mesh of a surface. The idea is to put into the scene a series of essentially transparent

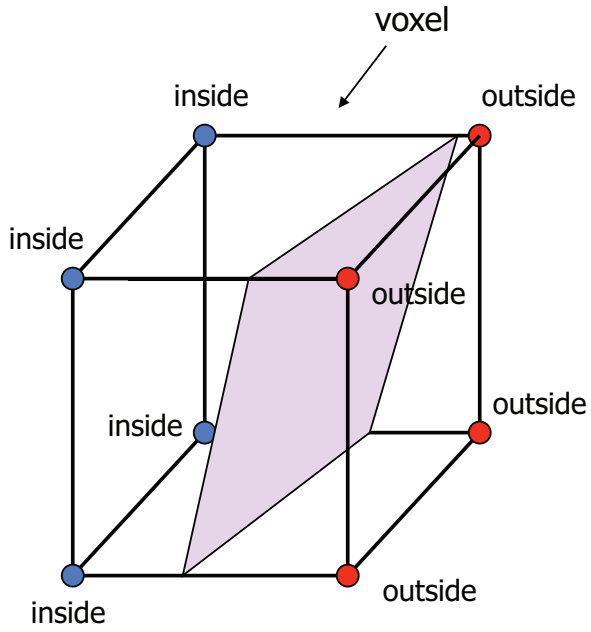


Figure 15: The values from 8 voxels creating a cube the left-hand side of which is 'inside' a surface, and the right-hand side of which is 'outside'. A single quadrilateral drawn on the horizontal edges at the appropriate iso-value creates part of the surface.

polygonal planes, and then to project onto these layers of the volume data, so that when you look 'through' the planes, the various different layers line up to give the impression of a soilid object (a bit like looking through several equally spaced panes of glass, with different bits of a picture painted on them to give the impression of depth). The basic idea is shown in Figure 19.

The first requirement here is to generate the proxy planes onto which the different parts of the volume will be drawn. This could be done easily by drawing quadrilaterals that are lined up with the world axes (as happens to be the case in Figure 19). This is fine if you know you are always going to be looking into the world perpendicular to the plans (i.e. through the 'front' of the world); but it's dreadful if you happen to position your viewpoint so that it looks 'side on' at all the proxy planes. Under these circumstances you'd not really see the composite volume image at all, since you'd be looking along the edges of the planes rather than directly into them.

The solution is to recreate the volume planes so that they are perpendicular to view vector—or put another way, parallel to the image plane, which means recalculating them every time the eyepoint moves. Figure 20 shows this effect; in each case the triangles forming the proxy geometry layers are arranged so as to be parallel to the image plane, regardless of the orientation of the cube. You can see that in the middle cube this creates nearly equilateral-looking triangles, whereas in the left and right cubes the triangles are more distored to keep the plane of the triangles aligned correctly.

Having created the proxy geometry (which is a fairly painless calculation really), it then remains to decide how many layers are necessary to give a good effect. This relies rather on the nature of the underlying data; the more rapidly or unpredictably the values change, the more layers will be needed to avoid the chances that important visual artefacts get lost in the gaps between proxy layers.

Once suitable layers have been drawn, a variety of approaches can be used to 'paint' onto them.

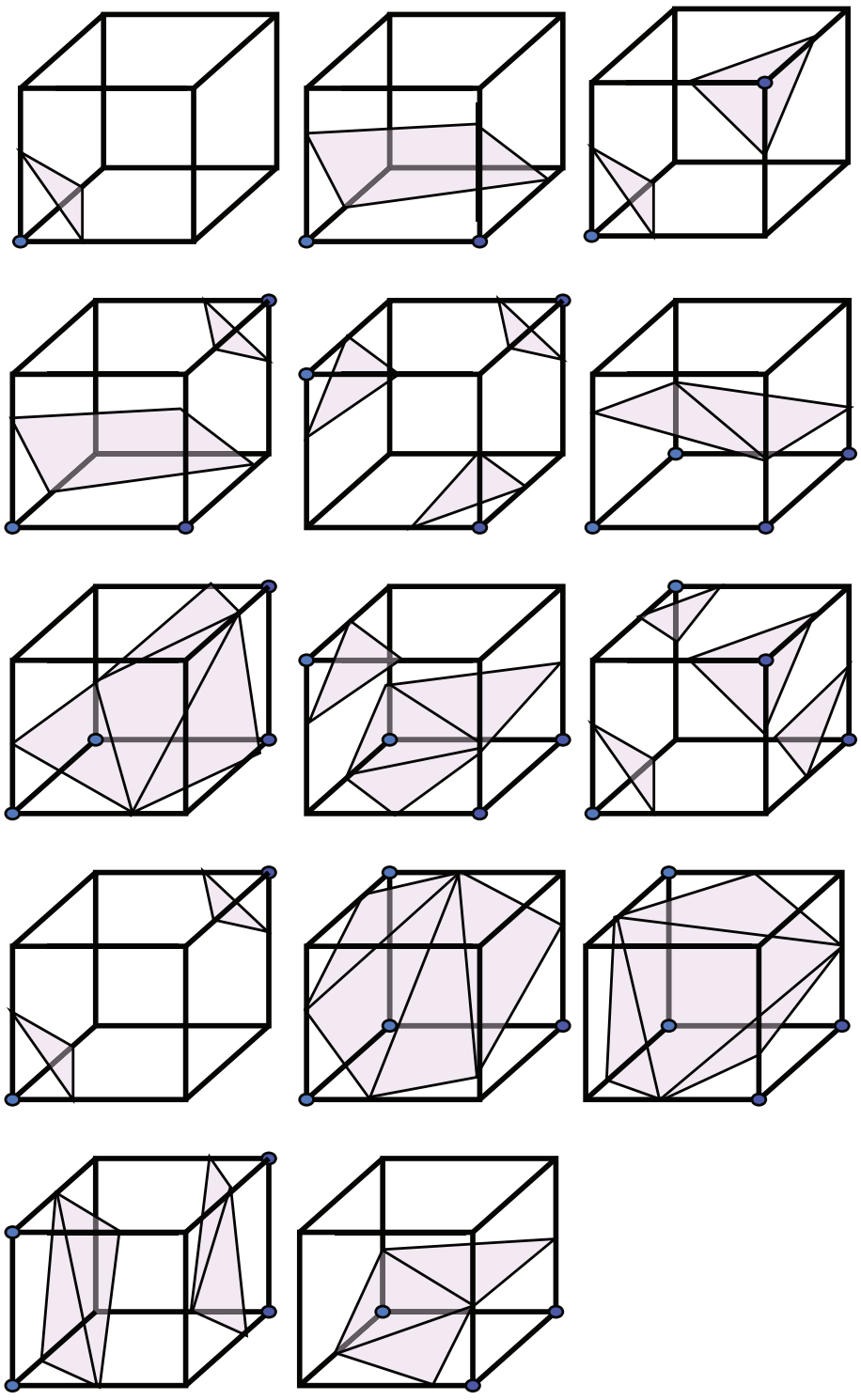


Figure 16: The 14 different variations of 'in' vertices (marked with a blue dot) and 'outside' (without a dot) that are used in the Marching Cubes algorithm.

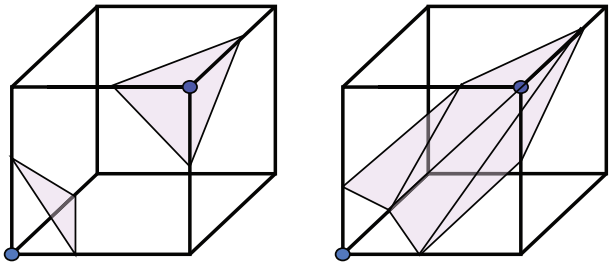


Figure 17: Two valid sets of polygons that would separate out the 'inside' vertices (marked with dots) from the 'outside'.

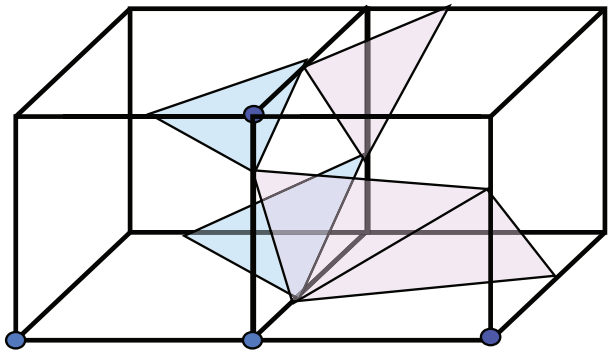


Figure 18: A discontinuous mesh created by the ambiguity shown in Figure 17.

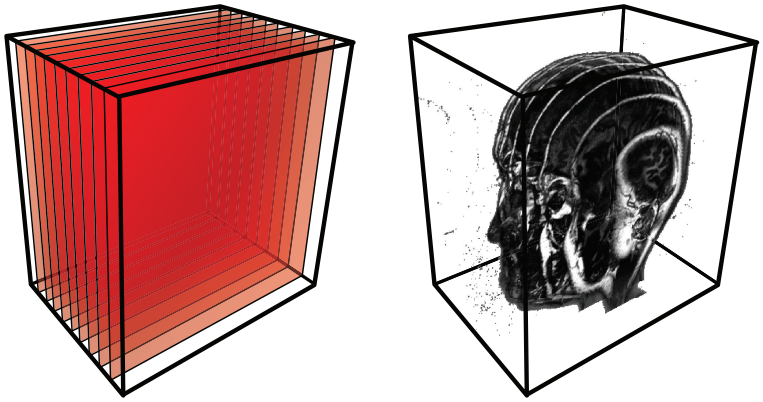


Figure 19: An example of axis-aligned proxy geometry (left), and volume data projected onto the proxy planes. Courtesy of Christof Rezk Salama, Klaus Engel and Markus Hadwiger.

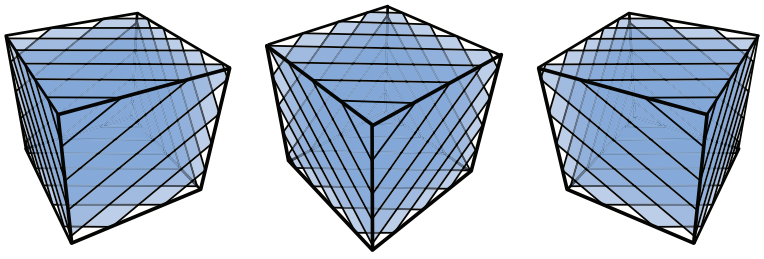


Figure 20: Three examples of viewplane-aligned proxy geometries. Courtesy of Christof Rezk Salama, Klaus Engel and Markus Hadwiger.

2.4.1 Splatting

*(This section and the two that following are taken from the Wikipedia entry on **Volume Rendering**[W], and is modified only slightly.)*

One of the early techniques in this area is called **splatting**[W]. This is a technique which trades quality for speed. Here, every volume element is splatted, as Lee Westover said, like a snow ball, on to the viewing surface in back to front order (Westover, 1991). These splats are rendered as disks whose properties (color and transparency) vary diametrically in normal (Gaussian) manner. Flat disks and those with other kinds of property distribution are also used depending on the application.

2.4.2 Shear warp

The shear warp approach to Volume Rendering was developed by Cameron and Undrill, popularized by Philippe Lacroute and Marc Levoy (Lacroute and Marc Levoy, 1994). In this technique, the viewing transformation is transformed such that the nearest face of the volume becomes axis aligned with an off-screen image buffer with a fixed scale of voxels to pixels. The volume is then rendered into this buffer using the far more favorable memory alignment and fixed scaling and blending factors. Once all slices of the volume have been rendered, the buffer is then warped into the desired orientation and scaled in the displayed image.

This technique is relatively fast in software at the cost of less accurate sampling and potentially worse image quality compared to ray casting. There is memory overhead for storing multiple copies of the volume, for the ability to have near axis aligned volumes. This overhead can be mitigated using run length encoding.

2.4.3 Texture mapping

Many 3D graphics systems use texture mapping to apply images, or textures, to geometric objects. Commodity PC graphics cards are fast at texturing and can efficiently render slices of a 3D volume, with real time interaction capabilities (see Figure 21). Workstation GPUs are even faster, and are the basis for much of the production volume visualization used in medical imaging, oil and gas, and other markets. In earlier years, dedicated 3D texture mapping systems were used on graphics systems such as Silicon Graphics InfiniteReality, HP Visualize FX graphics accelerator, and others. This technique was first described by Bill Hibbard and Dave Santek (Hibbard and Santek, 1989).

These slices can either be aligned with the volume and rendered at an angle to the viewer, or aligned with the viewing plane and sampled from unaligned slices through the volume. Graphics hardware support for 3D textures is needed for the second technique.

Volume aligned texturing produces images of reasonable quality, though there is often a noticeable transition when the volume is rotated.

2.4.4 GPU-accelerated Volume Rendering

A recently exploited technique to accelerate traditional Volume Rendering algorithms such as ray-casting is the use of modern graphics cards (as an example see (Engel, Kraus, and Ertl, 2001), though there are many others). Starting with the programmable pixel shaders, people recognized the power of parallel operations on multiple pixels and began to perform general-purpose computing on (the) graphics processing units (GPGPU). The pixel shaders are able to read and write randomly from video memory and perform some basic mathematical and logical calculations. These SIMD processors were used to perform general calculations such as rendering polygons and signal processing. In recent GPU generations, the pixel shaders

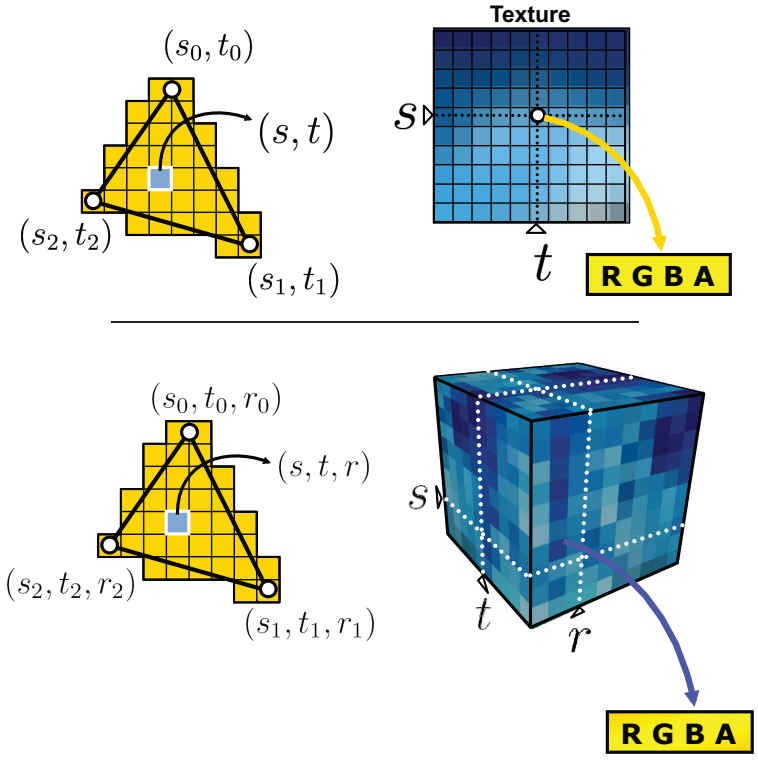


Figure 21: 2D and 3D texture mapping. Courtesy of Christof Rezk Salama, Klaus Engel and Markus Hadwiger.

now are able to function as MIMD processors (now able to independently branch) utilizing up to 1 GB of texture memory with floating point formats. With such power, virtually any algorithm with steps that can be performed in parallel, such as volume ray casting or tomographic reconstruction, can be performed with tremendous acceleration. The programmable pixel shaders can be used to simulate variations in the characteristics of lighting, shadow, reflection, emissive color and so forth. Such simulations can be written using high level shading languages.

2.5 A comparison of direct and indirect techniques

As should be obvious, the Direct Volume Rendering approach, rather like basic Ray Tracing, creates a rendered image on the viewplane, which must be recalculated if the position of the viewpoint changes, or if you decide to change the opacity/colour mappings; it's therefore not a hugely attractive approach for interactive applications (though, again like Ray Tracing, the value for each pixel is independent of every other, so it's once more an embarrassingly parallel problem, amenable to concurrent computation solutions). However, one of the significant advantages of DVR is that it does not make any 'hard and fast' decisions about the underlying data—there are no 'hard' cutoffs involved, so less chance of visual artefacts being created or missed.

Indirect techniques, on the other hand, end up creating polygonal representations of the data (whether as an iso-surface mesh, or by projection onto proxy geometry). Although these create representations that are viewpoint independent (so the can be rendered in realtime using normal OpenGL-like techniques), there is a much greater chance of false positive or false negative artefacts (e.g. if you choose the wrong value to create an isosurface from, or have an insufficient number of proxy-planes, then important features of the data can be missed in the

rendering).

References

Engel, Klaus, Martin Kraus, and Thomas Ertl (2001). "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading". In: *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS workshop on Graphics hardware*. HWWS '01. Los Angeles, California, United States: ACM, pp. 9–16. ISBN: 1-58113-407-X. DOI: 10.1145/383507.383515.

Hibbard, William and David Santek (1989). "Interactivity is the key". In: *Proceedings of the 1989 Chapel Hill workshop on Volume visualization*. VVS '89. Chapel Hill, North Carolina, United States: ACM, pp. 39–43. DOI: 10.1145/329129.329356.

Lacroute, Philippe and Marc Levoy (1994). "Fast volume rendering using a shear-warp factorization of the viewing transformation". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. SIGGRAPH '94. New York, NY, USA: ACM, pp. 451–458. ISBN: 0-89791-667-0. DOI: 10.1145/192161.192283.

Levoy, M. (1988). "Display of surfaces from volume data". In: *Computer Graphics and Applications, IEEE* 8.3, pp. 29–37. ISSN: 0272-1716. DOI: 10.1109/38.511.

Lorensen, William E. and Harvey E. Cline (1987). "Marching cubes: A high resolution 3D surface construction algorithm". In: *SIGGRAPH Comput. Graph.* 21.4, pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422.

Westover, Lee Alan (1991). "Splatting: a parallel, feed-forward volume rendering algorithm". UMI Order No. GAX92-08005. PhD thesis. Chapel Hill, NC, USA.