# A Concise Review of 3D Technology

## *What We Mean by What We Say About What 3D Chips Do*

*by Peter N. Glaskowsky*

The microprocessor industry has buzzwords for everything. So does the 3D industry. Unfortunately for readers of this newsletter, there's little overlap between the two sets, and some of the terms don't mean quite the same thing in both contexts. In our coverage of microprocessors and 3D chips, we make free use of the appropriate buzzwords, but we often don't stop to define many of the terms that are unique to 3D chips.

What follows is a modest attempt to rectify this omission and, we hope, to provide advance warning of some of the 3D buzzwords we'll be using in the future. We'll also identify the industry leaders in some of the key elements of 3D-chip design.

To make this glossary more useful, we've organized it according to the order these terms appear in a description of the 3D pipeline. We begin, therefore, by defining that.

### The 3D Pipeline

Broadly, the **3D pipeline** comprises the sequence of steps required to **render**, or generate an image of, a 3D **scene**. The target of the rendering operation is a **frame buffer**, a 2D array of picture elements (**pixels**) to be displayed on the computer monitor. Pixels usually consist of red, green, and blue **color values**, each with five to eight bits of precision.

Figure 1 shows a sample 3D pipeline. These steps may be performed in different orders depending on the implementation, but all 3D pipelines begin with scene definition.

Scenes are defined by a data structure known as the **scene database**. This database provides a mathematical representation of the virtual **objects** in the scene and their position relative to each other. The application must also define the position of the **view point**—a virtual camera—relative to the scene. The process of updating these definitions as objects move or change is called **scene management**.

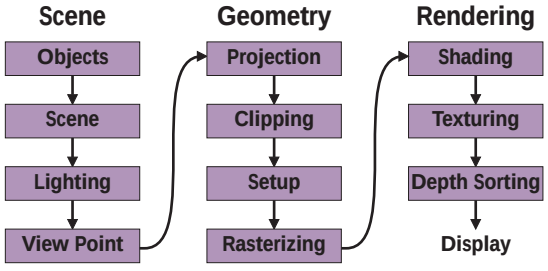


**Figure 1.** A representative 3D pipeline comprises scene-definition, geometry, and rendering operations.

In today's 3D programs, **object models** are usually defined by a set of **polygons**—typically triangles or quadrilaterals—that represent the surface of the object. The polygons themselves are defined by the positions of each **vertex** in three dimensions. Figure 2 shows how **polygon strips** and **meshes** can be built from lists of vertices, reducing the number of distinct vertices per polygon in an object model.

These polygonal definitions require a very large number of polygons to describe smooth curved surfaces, and they cannot effectively define some objects (such as hair, fire, and clouds of dust) at all. Also, they say nothing about the interior of an object.

Several methods have been developed to solve these problems. Mathematical functions such as nonuniform rational B-splines (**NURBS**) and **quadratic patches** can describe curved surfaces directly and are more compact than long lists of polygons. We expect to see 3D systems capable of rendering directly from these functions within the next few years, but most of today's 3D engines require polygons. Curved-surface models are converted to polygonal models by a process known as **tessellation**. This conversion may take place just before an object is rendered, in which case the number and size of the polygons being generated may be adjusted according to the size of the object on the screen—visibly smaller objects need fewer polygons. When tessellation is performed in this way, it is described as **progressive** or **multiresolution meshing**, or **mesh refinement**.

Three-dimensional arrays of **voxels**—values that describe the contents of a volume—are used by **volume rendering** systems such as Mitsubishi's VolumePro (see MPR 11/16/98, p. 22) to display the cross-sections of solid objects, a capability lacking in polygonal 3D systems.

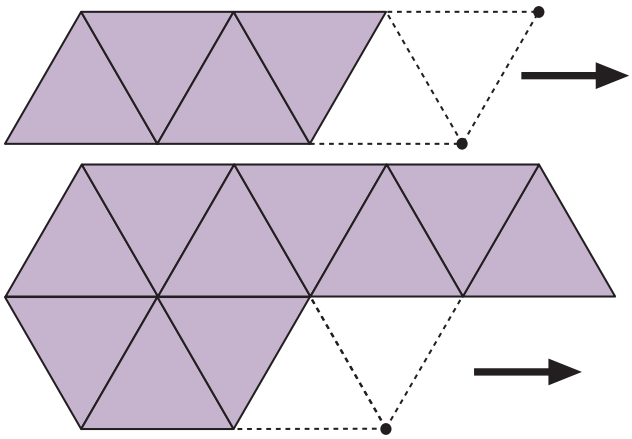

**Figure 2.** Polygon strips and meshes can be built up from a list of vertex coordinates to reduce the size of object models.

## Geometry Processing

Each 3D object model in the scene has an independent coordinate system, or model **space**, aka **local space**. The origin in a model of a monster for a 3D game may be located near its left foot, while the origin for the scene in which the monster appears will likely be elsewhere. If the monster turns or falls over, its X, Y, and Z axes will no longer be aligned with those of the scene, which has its own **scene space** or **world space**.

All of the independent coordinate systems used within the scene must eventually be converted into a single coordinate system related to the view point: the **view space**. Matrix operations are used to **transform** one coordinate system to another.

Any lighting effects used in the scene require a similar process of coordinate transformation. There are many different kinds of lights, from omnidirectional lights to tightly focused spotlights, and each requires a different form of processing. Lighting calculations can be simplified by assuming that the light is very far away, in which case every lit object in the scene receives a similar amount of illumination. If the distance is relatively small, the inverse-square law must be considered in the calculations.

Transform and lighting calculations are part of **geometry processing**, the first step in 3D rendering. Geometry processing is performed by the host processor in today's mainstream 3D systems, but most professional 3D add-in boards provide **hardware geometry acceleration**. Several leading makers of PC graphics chips have announced plans to integrate geometry acceleration into next-generation 3D chips.

Once relieved of these repetitive geometry calculations, the host processor can perform more valuable work. Both games and professional applications will benefit from **physics-based modeling** of object behavior. Objects that fall, collide, bounce, and deform like real objects will make car crashes in racing games more entertaining and make forensic car-accident simulations more accurate. Monsters and other computer-generated game characters will also become more interesting when more CPU time can be devoted to behavior simulation and other artificial-intelligence algorithms.

## Setup Processing

Once a consistent set of 3D vertex coordinates in view space has been generated, the next step in the 3D pipeline is to convert these coordinates to **screen space**: the 2D coordinates of pixels on the display device plus a third coordinate that defines the distance from the view point.

Translating from the view space to the screen space is relatively simple, since these two coordinate systems are mathematically related. This final translation is known as **setup processing** and is performed in hardware on almost all of today's PC graphics chips.

These calculations are performed with a resolution greater than the X and Y pixel coordinates of the display itself to minimize accumulated error. The edge of a polygon bears no particular relationship to the edges between pixels, so setup processing should preserve **sub-pixel accuracy** on vertex coordinates. The setup accuracy of graphics chips varies widely, from as few as one extra bit to as many as eight.

The final step of setup is **edge walking**, also known as **scan-line conversion**. In this step, the setup engine generates pairs of X coordinates that identify the leftmost and rightmost edges of the polygon for each **scan line** the polygon touches on the display. These X coordinates are accompanied by the distance coordinate, which is used later in the pipeline.

## Culling and Clipping

At various points during the geometry and setup operations it becomes possible to identify polygons that will not be visible on the screen. **Culling** such polygons as early as possible reduces the work that must be done by later stages in the pipeline. Some culling can even be done by the 3D application itself, which may know that some elements of the scene database are not visible from the current view point. This is known as **database culling**. For example, if the view point is in one room of a house, objects in most of the other rooms can be culled out immediately.

Culling within the rendering pipeline itself usually begins by rejecting polygons that face away from the view point—as many as half of the total. Since these represent the far side of objects, they will not be visible—they will be occluded by the near side of the object.

Object definitions may include **bounding boxes**, simple 3D shapes that encompass the entire object. Bounding boxes can be transformed to the view space and tested against the visible limits of the view—the **view port**—without transforming every polygon in the object. If the box is entirely invisible, the whole object may immediately be culled. Only if the box is partially visible must the object within be subjected to polygon-by-polygon checks.

Any polygon that intersects the edge of the view port will have the offending portion clipped off by the setup engine, which will not generate coordinates outside the range supported by the display device.

## Depth Sorting

Culling and clipping don't catch all of the invisible polygons. Before each pixel across the specified scan-line segment can be drawn on the screen, the rendering engine must check to verify that the polygon is actually visible at that pixel. A previously drawn polygon may overlap or intersect the current polygon, making some or all of the current polygon invisible.

Early 3D systems used the **Painter's Algorithm** to avoid this problem, drawing polygons in order, according to their distance from the view point. The most distant polygon is drawn first, with the nearest polygon coming last, ensuring that every polygon is visible. The Painter's Algorithm requires that the entire polygon list be sorted prior to rendering, imposing a substantial burden on storage and computation. It also requires polygons to be drawn, even if they will not be seen.

The crucial problem with the Painter's Algorithm is that it cannot handle intersecting polygons, as Figure 3 shows. The potential for such polygons in complex 3D scenes makes this algorithm impractical.

Today, rendering engines use the distance information passed by the setup engine to determine whether the current pixel on the current polygon is closer or more distant than the last polygon drawn at that location. The distance to the last polygon for each pixel on the screen is stored in a separate array with the same X and Y size as the screen. (This buffer is initialized to the maximum-distance value prior to rendering each image.) When the values are proportional to the distance, the values are called **Z values** and the array is called a **Z buffer** or **depth buffer**.

The number of overlapping polygons in the scene (either the average or maximum number, depending on the context) is called the scene's **depth complexity**. Some 3D engines, notably those based on the VideoLogic/NEC PowerVR architecture (see MPR 6/23/97, p. 1), do depth sorting for the entire scene prior to rendering. These engines avoid rendering invisible polygons and therefore achieve their best performance relative to conventional 3D chips on scenes with a high depth complexity.

## Shading

Once the rendering engine knows that the current polygon is visible at the current pixel, it must calculate the pixel's new color. If the polygon is part of an object with a specific color, the polygon's vertex definitions include color values—a technique called **per-vertex color**. When the colors at each vertex of a polygon are different, the color of each pixel within the polygon is usually calculated by a 2D interpolation across the polygon face.

Lighting effects may modify the pixel's calculated color. Lighting calculations may be made at a single point for each polygon (known as **flat shading**), or made at each vertex and interpolated across the polygon face in a technique known as **Gouraud shading**, or calculated independently for each pixel (**Phong shading**). Phong shading produces the best results but requires the most work. No mainstream 3D chips use Phong shading today, but some specialty and high-end 3D engines can use Phong shading, and we expect this feature to become widely supported over the next few years.

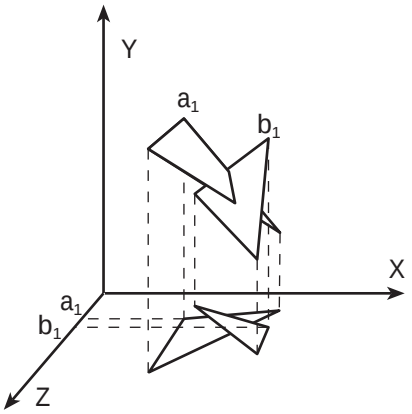


**Figure 3.** Two intersecting polygons cannot be drawn correctly using only depth sorting, because neither is entirely in front of the other.

Today's 3D chips use simple red-green-blue (**RGB**) color models. In these models, a green object illuminated by a red light may not reflect any of the light and thus appear to be black. In real life, a monochromatic green object would indeed look black when illuminated solely by a pure red light, as from a laser. However, real life contains few monochromatic objects, and lasers are rarely used to illuminate real-life scenes. Interactions of real objects and real light sources are based on the complete spectrum of visible light (not to mention fluorescence effects). Future 3D hardware will adopt more realistic color and lighting models to produce more natural visual effects. For example, more than three colors could be used, or colors could be represented by equations that define intensity as a function of wavelength.

Object definitions may also specify information about the object's reflectivity. Shiny objects produce what are known as **specular** reflections; dull objects produce **diffuse** reflections. The appropriate use of specular and diffuse lighting effects—and chips that implement them correctly—can be very effective in making objects look more realistic. A human face exhibits regions of highly specular reflection in the eyes, and very diffuse reflections on the cheeks. Areas such as the end of the nose have an intermediate appearance.

In the real world, lights reflect off objects to illuminate other objects; even the deepest shadow is not completely black. A complete and accurate model of these reflections would require simulating the path of every photon from every light source in the scene—clearly an impractical approach. Even a partial simulation produces good results, however, and this is the basis of **radiosity lighting**. Radiosity models are built by an iterative process. In each iteration, the lights visible from each object, including reflected light, are used to recalculate each object's illumination. The visible results improve rapidly for the first few iterations, then the results begin to converge. Even so, radiosity lighting can be prohibitively compute-intensive when the objects in the scene, or especially the original light sources, are moving around. Radiosity lighting is not supported by any mainstream 3D chip today, but it represents another opportunity for progress and differentiation in future chips.

The best results are achieved by **ray tracing**, an algorithm that traces the path of the photons that would reach each pixel on the screen. These paths are traced in reverse from the view point all the way back to the light sources that issued the virtual photons. Ray tracing can require even more calculations than radiosity lighting. Only one company, Advanced Rendering Technology (*www.art.co.uk*), makes ray-tracing accelerator chips.

## Texture Mapping

Colors and lights alone produce good visual results for synthetic objects. Simple shading also works for some real-world objects composed of materials such as plastic and steel. Most real-world objects have a much more complex appearance, however.

**Texture mapping** wraps 2D images around the surfaces of 3D objects to produce more natural results. These images, known as **texture maps**, add a great deal of detail to the object without adding any additional polygons. Texture maps may also be used to simulate reflections from shiny objects, a technique known as **environment mapping**.

Texture mapping begins with texture coordinates associated with each polygon vertex. These coordinates, defined when the object was created, tell the 3D engine what part of the texture map should be applied to each polygon. More than one texture may be applied to a single polygon to simulate lighting effects or multiple independent textures in a process known as **multitexturing**.

The texture-mapping unit in the rendering engine uses the texture coordinates to calculate the precise point in the texture map of the center of each pixel to be rendered. Optionally, the rendering engine may select from a set of texture maps of varying resolutions to find the closest match to the displayed resolution of the object, which declines as the apparent distance to the object increases. Multiple texture maps stored together to support this method are called **MIP maps**, for the Latin phrase *multum in parvo*, or "much in little." Each texture map in a MIP map is called a **level**, since each has a different **level of detail**.

There are many different algorithms for texture mapping. **Point sampling** takes the texture pixel, or **texel**, that lies closest to the calculated point and uses that texel's color for the screen pixel (modified by lighting effects, if any). This algorithm tends to produce poor results, since a single texel may be assigned to two adjacent screen pixels, while a nearby texel is skipped entirely.

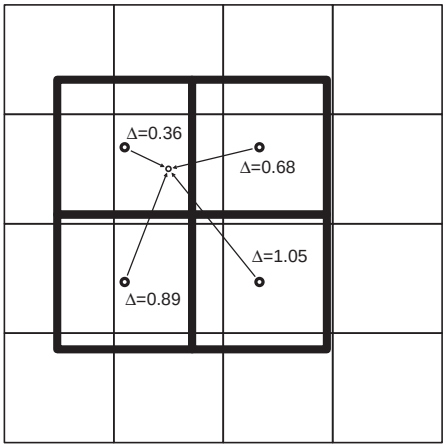


**Figure 4.** Bilinear filtering uses a weighted average of the nearest four texel values to calculate the texture of the current pixel.

**Bilinear filtering**, shown in Figure 4, uses a weighted average of the four nearest texels from a single texture map to calculate the pixel color. This method produces much better results than point sampling, but it requires four times the effective bandwidth to texture-map storage. The primary problem with bilinear filtering comes at boundaries between MIP-map levels, where a visible edge is created.

When bilinear filtering is performed on the texel data from the two MIP-map levels with the closest effective resolution and the results are blended using another weighted average, the visible boundaries between MIP-map levels are eliminated. This technique is known as **trilinear filtering**. It is the best method in common use today but requires twice the texture bandwidth of bilinear filtering.

Trilinear filtering can be further improved by changing the shape of the filter window according to the effective "footprint" of the screen pixel on each texture map. Figure 5 illustrates this technique, called **anisotropic filtering**. For an object surface at a 66° angle to the view point, a typical anisotropic filter uses 10 texture samples from each MIP-map level and requires five times the bandwidth and computation needed for bilinear filtering.

Texture maps may also be used to describe changes in the location of each pixel in a polygon. Texture maps that contain relative-distance information—displacements above or below the plane of the polygon—are called **displacement maps**, **bump maps**, or **perturbation maps**. Such maps are used to modify lighting calculations, just as bumps and dents in a real object produce brighter and darker regions on the object's surface.

## Blending

Color, texture, and lighting values are eventually combined in **blending**, the final stage of rendering. One last possible problem must be handled at this point—polygon edges that cover only part of a pixel. The simplest solution is to compute the fraction of the pixel covered by the polygon and blend that fraction of the new color with the old color, but this leads to errors. The best results are achieved by keeping track of all these pixel fragments and combining them intelligently at the end of the rendering process.

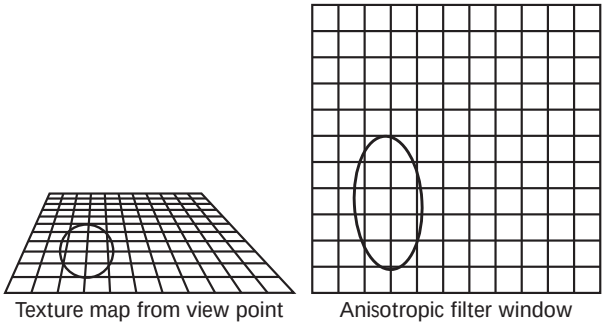


**Figure 5.** Anisotropic texture filtering uses all the texels covered by the current screen pixel to compute a more accurate result.

The more common solution is to render the image at a higher resolution than the screen can display (typically two to four times higher linear resolution). This high-resolution result is then filtered down to the screen resolution, eliminating most visible rendering errors. This process is known as **antialiasing** and is supported by the best new graphics chips. Antialiasing typically requires 4 to 16 times as much frame-buffer capacity and bandwidth as nonantialiased rendering.

Rendering translucent objects can add a great deal of complexity to the pipeline. When the closer of two objects is translucent, its color must be mixed with that of the more distant object. This mixing is controlled by the translucency factor, known as the **alpha value**. Alpha values, usually denoted by $\alpha$, are kept with the R, G, and B color values in vertex and texture-map definitions.

The results of the blending operation represent the final color of the pixel, unless it is overwritten by a subsequent polygon. Once all polygons have been drawn, the new screen image is ready to be displayed. Ordinarily, the 3D subsystem will have two frame buffers available, a configuration known as **double buffering**. After the scene is fully rendered into one frame buffer, that buffer is displayed on the monitor while the other is used for rendering. The buffer being displayed is known as the **front buffer**, while the one being rendered into is called the **back buffer**.

Double buffering hides the partially rendered scene from the viewer, but it can cause a pipeline stall if the back buffer is fully rendered before the front buffer has been fully displayed. Additional buffers can be used to reduce the likelihood of this type of stall.

When the back buffer is fully rendered, the front buffer has been fully displayed, and the display device is ready to display another page, the functions of the two buffers are swapped—an operation known as a **page flip**. Rendering resumes into the new back buffer while the new front buffer is displayed. (The same Z buffer is used for each rendering pass, since the Z buffer is not needed to display the image.) This sequence is repeated as rapidly as possible. **Frame rates** above about 20 frames per second (fps) produce an illusion of smooth motion; this effect improves with higher frame rates to a viewer-dependent threshold of about 80 fps.

Today's 3D chips can already hit 80 fps on 3D scenes composed of a few thousand polygons when displayed on a medium-resolution monitor (about $800 \times 600$ pixels). Future performance improvements will go to increase polygon count, monitor resolution, and image quality.

To match the capabilities of the human visual system, the performance of today's 3D chips will need to improve by about seven orders of magnitude—at least 12 years of progress, even at today's amazing pace of 4× per year. Today's 3D architectures probably can't be extended that far; new techniques, and a lot of engineering effort, will be needed to reach this goal. M

*This article, with links to relevant books and chip-vendor Web sites, is online at* www.MDRonline.com/3d.