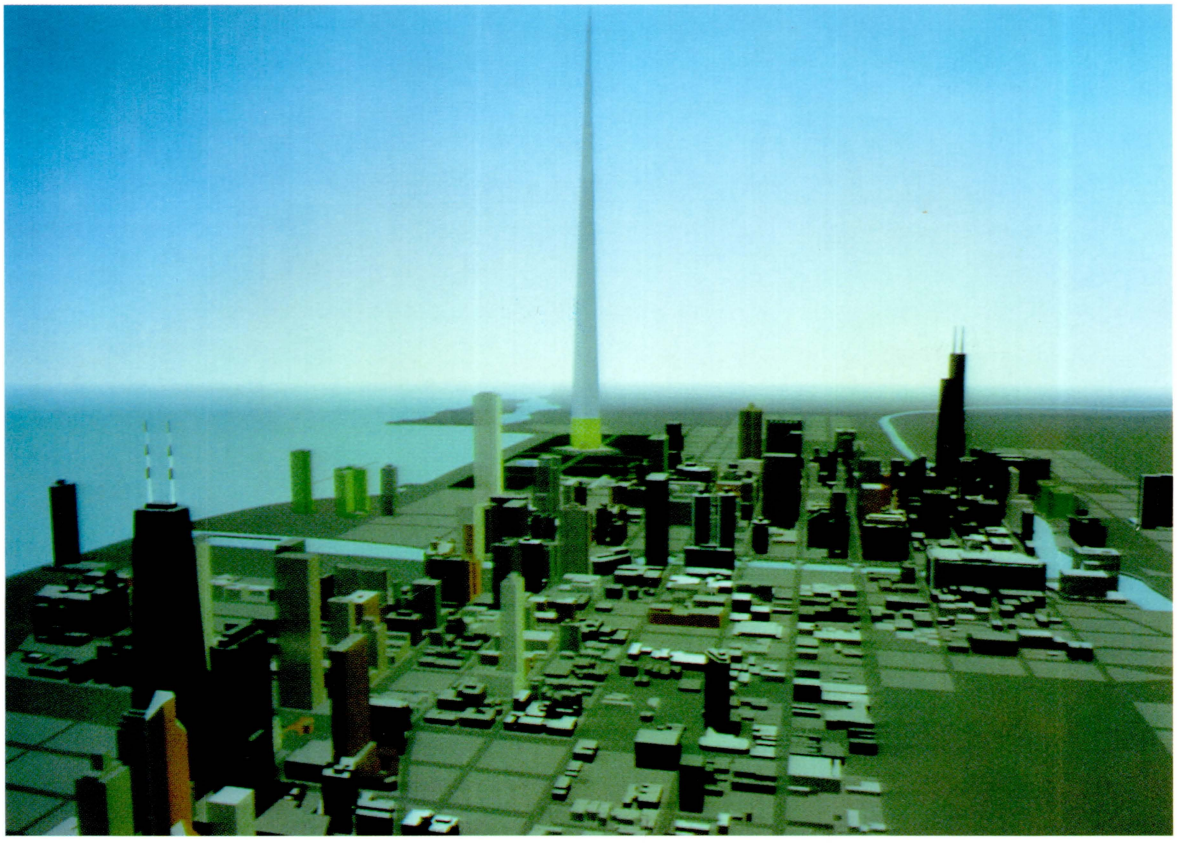
CONVEX Technical Paper Series

Supercomputer Rendering

Scott Dyer
The Ohio Supercomputer Project
The Ohio State University



This image of Chicago was created using data provided by the architectural firm of Skidmore, Owings, and Merrill. The building in the center of the image is a one-mile-high structure designed by Frank Lloyd Wright in 1955. This image contains nearly 60,000 polygons and took about 25 minutes to calculate

Abstract

The computational needs of computer graphics have always outstripped available resources. With individual images often requiring an hour or more to compute, and animated sequences comprising several thousand frames, the need for high-speed computation and efficient implementation is great. Until recently, the enormous computing requirements of computer animation have been met either with expensive, monolithic machines or with networked groups of smaller machines. The advent of affordable vector processors provides an opportunity to significantly reduce the time and cost required to calculate complex images.

After a general description of the process of generating high-quality imagery, two specific rendering algorithms are presented. These algorithms represent two differing approaches to image computation on vector processors. The first is a traditional scan-line technique, and the second randomly samples the image in an approximation of the operation of the human visual system. The methods used to vectorize each algorithm are discussed, as is the application of vectorization to the general problems of computer graphics.

Introduction

The human visual system is enormously complex and powerful. We think nothing of being able to recognize a friend's face in a crowd, of reading even the worst handwriting, or of noticing the single item out of place in our crowded living rooms. The processing power of the human visual system cannot be approximated by even the most expensive computer systems and may never be equalled.

At the same time, we live in a minutely detailed environment. A tree, for example, is not simply a stick topped with a green blob, but is instead a complex, twisted shape covered by textured bark and bearing hundreds of thousands of uniquely shaped and colored leaves.

Attempting to simulate reality with a computer image seems quite hopeless in the face of all this. Not only is the world so detailed and complex that no computer would be capable of representing even a single tree, but the human visual system is so powerful that missing detail is noticed immediately.

There is hope, however. Remarkable realistic imagery can be generated by computer, but the task often involves approximation, trickery, and wholesale deceit. In computer graphics, the key is how the final picture looks, not whether it is scientifically correct. Even with approximation, however, the computing needs of realistic imagery are immense. It is common for complex images to require hours or days to compute on the largest scalar machines. Vector architectures provide a unique opportunity to reduce the cost of realistic computer pictures.

Realistic computer imagery serves many needs. Designers can view their creations without the expense of creating a physical model. Architects can design a new building, experiment with materials, and even view the building in the context of its surroundings. Medical researchers can preview operations and examine complex internal structures. The entertainment industry can transport viewers to unseen, even unreal, worlds through realistic computer imagery.

This paper addresses the process of creating realistic computer pictures. The application of vector processing to the problems of computer image generation is discussed in detail.

Rendering

Computer images can be as simple as a line drawing or as complex as images that approximate the realism of photographs. Depending upon the relative complexity of the image, the time required for average computers can calculate it ranges form near real time (less than one-thirtieth of a second) to several days. This paper will consider only the generation of high-quality computer imagery that includes realistic surface shading. These computer pictures are made in three stages: the definition of data, the description of the scene, and the rendering of the image. While this paper is primarily concerned with the latter stage, some discussion of the first two provides a better foundation for the analysis of rendering.

This three-stage process applies to conventional image making and computer graphics. For example, consider an architect who needs to draw a picture of a building. First, he must design the building; in this stage, he might examine blueprints or build a model. He is mentally constructing the data and relationships that define the building. In the second stage, he decides where the viewer will stand in his drawing. Will he draw the building from the front or the side? Finally, he will actually sketch and color the building. In this process, he converts his mental database and knowledge of perspective into a realistic drawing. He may include the effect of the sun and other lights in his image. The resulting picture is an effort to realistically portray his mental image of the building.

The generation of computer images follows these same three steps. The data definition stage creates a mathematical representation for the objects that will be visible in the final image. The scene description stage defines the position of the viewer, the positions of the lights that will shine on the objects, and other parameters that relate to the image as a whole. Finally, the rendering process converts the information from the first two stages into an image.

The definition of the data, the first stage in image making, can take several common forms. All common graphical data forms use a coordinate system to model the shape and position of the object in space. One of the simplest descriptions of an object uses many flat faces, called polygons, to approximate its shape and curvature. Representing a cube with this technique is obvious; representing a sphere is somewhat more difficult and is usually done by approximation. An object can be composed of many thousands of polygons. Each polygon is defined by a series of positions in space that form its vertices. Even complex objects can be described this way. For example, a cube has a total of six faces, while the skull (shown in Figure 1) has over 2000 faces.

By definition, polygons must be planar and convex. These constraints are primarily important in the determination of the *surface normal*. The surface normal is a vector-of-unit length that is perpendicular to the plane defined by the polygon. For a convex, planar polygon, the surface normal is the vector cross product of two of the edges of the polygon. The surface normal defines the shape of the object and determines shading in the rendering stage of the image making process. Nonplanar polygons do not

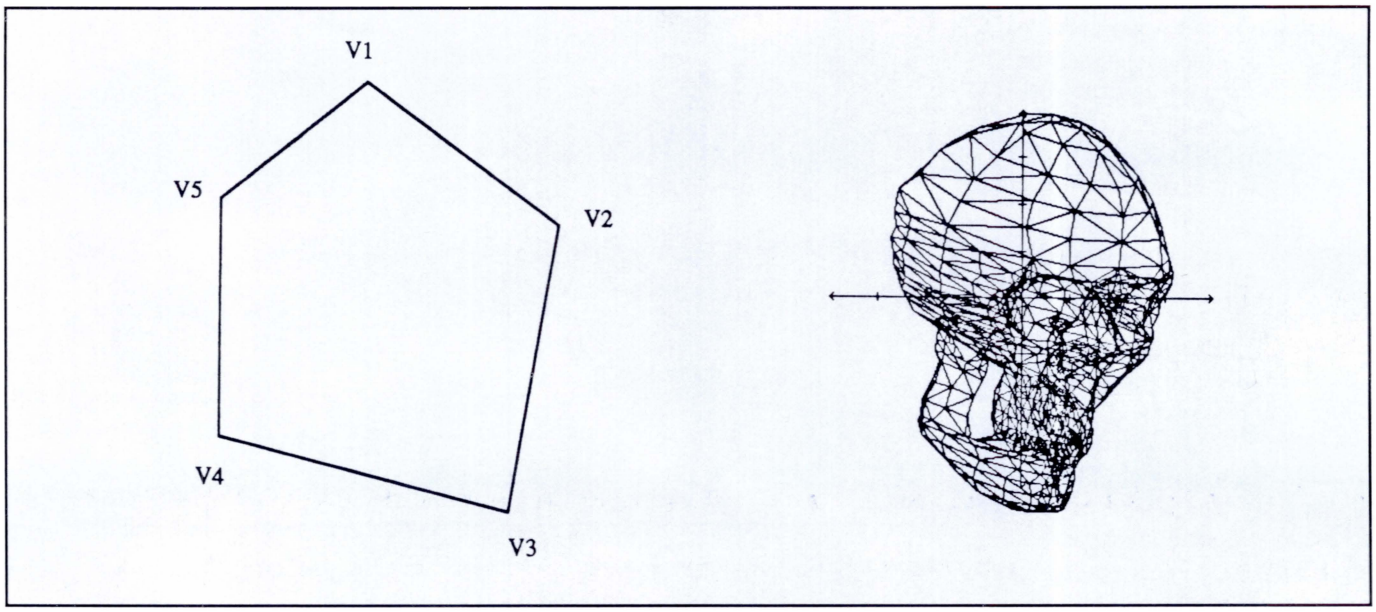


Figure 1: Polygonal data representation shows that the polygon has 5 vertices while the skull has 1919 vertices and 2084 polygons.

have a consistent surface normal. Concave polygons are difficult both to manipulate and to fill on the screen, and they reverse pointing surface normals depending upon which edges are used in the computation. Often, as is the case in the second rendering algorithm presented here, triangles are the only type of polygon permitted, since they are always both convex and planar.

The polygonal mesh determines the shape and structure of the object. Another important part of the data determines the appearance of the object by describing its coloring and surface quality. Together, the polygonal mesh and surface/color description comprise the complete data definition.

The second stage of the process to create an image is the description of the scene. Objects musts be positioned in space, and a viewpoint must be chosen. Each object is generally defined in its own coordinate system, with the origin at a convenient point. This coordinate system is called the *object* coordinate system. When objects are manipulated together to form a scene (by rotation, translation, and scaling), the final result is called the *world* coordinate system.

Finally, the position of the viewer is chosen, as well as the focal length of the lens used to view the scene. Lights are positioned to illuminate objects in the scene. These parameters, together with the object data described above, constitute a complete description of a scene.

The process of rendering, the final stage in the generation of computer images, converts this polygonal database and scene description into a picture. The rendering operation can be divided into four basic stages: transformation, hidden-surface removal, polygon tiling, and shading.

The transformation process converts the scene from the world coordinate system to the *screen* coordinate system. Raster display devices are organized like a grid with a number of vertical *scan lines*, each of which consists of a number of picture elements or *pixels*. A television standard video image, for example, is 640 pixels by 484 scan lines. This makes for a total of about 310,000 individual pixels on the screen. The screen coordinate system uses pixel and scan-line numbers to reference location. A perspective projection converts between the three-dimensional world coordinate system and the two-dimensional screen coordinate system. This creates an effect similar to that of photographic systems and the human visual system. Perspective foreshortening causes the projected size of objects to vary inversely with their distance from the center of projection (in this case, the position of the viewer).

Hidden-surface removal, the second stage in rendering, is the process of culling from the resulting perspective view those surfaces that are behind other surfaces (i.e., those that are invisible from the viewer's perspective). A great variety of hidden-surface algorithms have been developed, ranging from extremely simplistic approaches that mimic the way artists paint to complex algorithms that solve the problem quickly and efficiently. The painter's algorithm, for example, removes hidden faces by sorting the polygons according to their depth into the screen, then painting those furthest away first and those closest last. In this way, close polygons obscure, or paint over, those that are far away, and hidden surfaces are invisible. The problem with the painter's algorithm is the amount of excess computation that is performed: even if it will eventually be hidden, each polygon is painted.

Other hidden-surface techniques divide the objects and polygons until no hidden-surface problem exists.

These "divide and conquer" algorithms subdivide the screen until each area can be solved easily. Scan-line techniques, which are bound inseparably to the polygon tiling process, slice polygons into segments corresponding to each scan line. These slices are then sorted according to depth and distance across the screen.

Polygon tiling, the third stage of the rendering process, is often called scan conversion because each polygon is made to fit the pixel and scan-line structure of the display screen. All pixels that lie inside the boundaries of each polygon are shaded appropriately. The efficiency of this step is important; simply testing each pixel on the screen against every polygon would result in extremely poor performance. Scan-line techniques use the *coherency* of polygons to reduce the testing phase. This simply means that a convex polygon will intersect a given scan line only once and that all pixels between the intersection of the left edge and right edge will be within the polygon.

Unfortunately, the polygon tiling process leads to a basic problem in computer graphics—aliasing. Aliasing refers to the tendency to undersample areas of high detail. This results in a variety of undesirable effects. Straight lines that appear jagged, details that disappear and reappear, and thin lines that appear as complex moire´ patterns are all examples of aliasing. Indeed, aliasing is a familiar sight on television; often, when a newscaster wears a thin-striped shirt, it appears to change color and pulse as he moves about the screen.

Sampling theorem states that the sampling rate must be twice the highest frequency present in the original signal; however, practical and computational limitations prevent sampling at those rates. Thus several techniques exist to reduce aliasing.

The obvious solution is to sample the image at the highest rate possible and then average the results to fit the display screen. However, this technique is extremely expensive, and while it reduces the aliasing problem, it does not eliminate it.

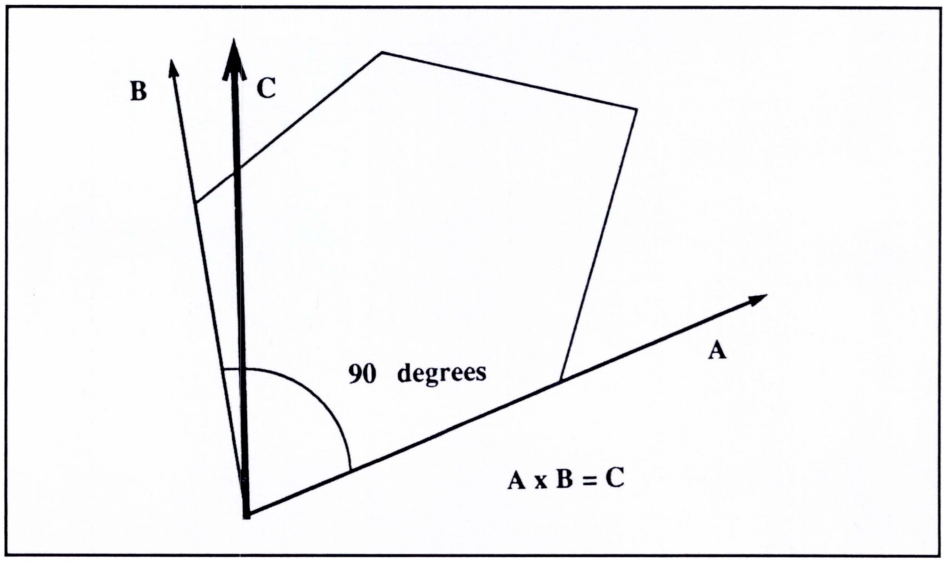


Figure 2: The polygonal surface normal.

Refinements of this technique only increase the sampling rate along edges and areas of high contrast. This reduces the time required to compute the image, but it has little effect on the final quality.

For scan conversion algorithms, a more common anti-aliasing technique treats individual pixels as finite areas rather than as single points. Rather than determine the color of an entire pixel by sampling the pixel at its center, polygons are clipped to fit the pixel boundaries. The area of the pixel covered by the polygon can be computed and used to determine the pixel's color. This approximates an extremely high sampling rate but avoids expensive recalculations for each pixel.

The shading process, the final stage in rendering, chooses colors for each pixel on the screen based on the properties of the materials being represented, the configuration of lights, and the position of the viewer. The simplest shaders model objects as perfect diffuse reflectors (such as a sheet of rubber or pieces of cloth) while the most complex use optical principles to represent the subtle effects of light.

Light that strikes an object is either absorbed, reflected, or transmitted. Absorbed light becomes heat and is for all purposes invisible. Reflected and transmitted light, on the other hand, make an object visible. Furthermore, which wavelengths of light are absorbed and which are reflected determine color. Reflected light can be either diffuse or specular. Diffuse reflection is light that is scattered equally in all directions, while specular reflection is highly directional light reflected from the surface of the object. A simple shading model includes both diffuse and specular light.

Lambert's cosine law relates the amount of diffusely reflected light to the cosine of the angle between the surface normal of the object and a vector to the light source. This light is independent of the position of the viewer and is usually implemented as the dot product of the surface normal and vector to the light source. Often, an ambient light "fudge" factor is added to avoid the sharp contrast between light and dark caused by the simple Lambert model. Without this addition, images tend to have a deep-space "Star Wars" look, because the parts of objects that face away from the light are completely black. While ambient light has no scientific analog, it represents the constant, nondirectional light present in most real environments. It is simply added to the Lambert cosine value and prevents an object from being completely dark on the sides that face away from the light source.

A realistic model for specular reflectioncalled phong shading was developed by Phong Bui-Tuong in 1975.

Specular light is approximated by a power of the cosine of the angle between the viewer and reflection vector. The surface normal is the bisector of the vector to the light source and the reflection vector.

Because a single surface normal is used to calculate the shade for an entire polygon, the resulting image appears to consist of many tiny facets rather than a smooth surface. While in some cases this is a desirable effect (such as a cube), surfaces are generally smooth and curved. To avoid faceting, an accurate normal must be approximated across the entire surface of the object. If a normal for each vertex in the object is computed by averaging the normals of the surrounding polygons, the resulting mesh of normals can be interpolated across the surface to simulate a smooth object. This technique is called *gouraud* shading.

Various rendering algorithms approach the four steps discussed above differently. Most share a common performance characteristic, however: the hidden-surface removal, polygon tiling, and shading steps tend to completely overwhelm the time required for the transformation stage. In many cases, the hidden-surface solution is inseparable from tiling and shading. Shading is expensive because it is done at least once for every pixel on the screen and because it generally involves both vector normalization (a square root) and a power function.

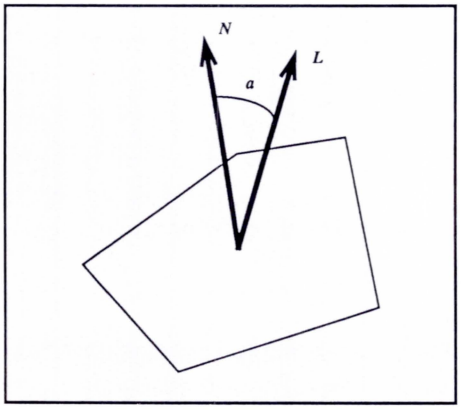


Figure 3: Lambert shading.

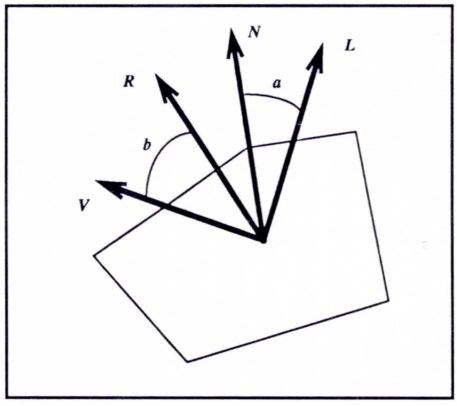


Figure 4: Phong shading.

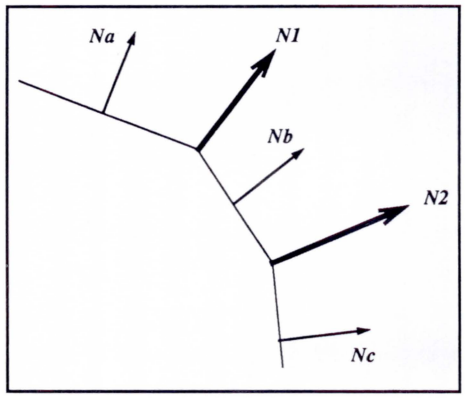


Figure 5: Gouraud surface normal shading (side view).

SCAN: A scan-line z buffer rendering algorithm

A z buffer rendering algorithm sorts all polygons that intersect a particular pixel and shades the polygons to those on the current scan line. The name z buffer refers to the state of the screen after the transformation step (stage one in the rendering process) where the x and y axes of the world coordinate system are aligned with the scan lines and pixels on the screen, and the z axis is projected into the face of the screen. The z depth of an object thus becomes a measure of its depth and is used to sort which polygons are closest to the viewer and therefore visible.

A significant portion of the scan-line z buffer algorithm is vectorized. Some parts of the process are immediate candidates for vectorization; for example, the transformation operations on the databases vectorize easily. However, these steps rarely account for more than five percent of the total computation time.

The calculation and normalization of the object surface normals can also be vectorized. It is the vectorization of the scan-line interpolation and shading steps, though, that can have the most significant impact on the execution speed of a scan-line z buffer algorithm.

Ordinarily, these three steps would not vectorize. As the algorithm proceeds down the screen, processing one scan line at a time, polygons are added to an active list on the first line in which they are visible and are deleted when they have been completely shown. Each active polygon is interpolated from the previous scan line to the current one, and all appropriate parameters (such as color and surface normal) are interpolated as well. This process proceeds across the scan line, activating polygons until the color for a particular pixel can be determined, and continues to the next line until the entire image has been rendered. The processes of hidden-surface removal, polygon tiling, and shading are all interleaved and inseparable. These steps do not easily vectorize because each polygon and pixel is handled individually.

On the other hand, by processing all active polygons for the current scan line simultaneously, the interpolation step can be vectorized. In the same manner, if a list of pixel fragments is composed, the shading step can be vectorized as well. Only the hidden-surface removal step is not vectorized. For convenience, each polygon is decomposed into a set of edge pairs that match left and right edges for easy interpolation. A complete description of the algorithm follows.

1. Set the defaults for the scene and object. Read the scene description and object.
2. Apply the world space transformation to the object.
3. Compute the surface normals for the object.
4. Apply the screen space trans-

form to the object.

5. Clip the remaining polygons to the viewing volume, interpolating values as necessary.
6. Compute the reflection vector and normalize the light.
7. Decompose the remaining polygons into edges and edge pairs. Since all polygons are convex, each right edge matches one left edge.
8. Calculate interpolation values (start, delta) for all parameters associated with the edges.
9. Insert each edge pair into the appropriate y bucket.
10. For each scan line:
    a. Add any new edge pairs that become active on this scan line to the active edge pair list.
    b. Interpolate the edge pairs from the top of the current scan line to the bottom, using the delta values computed above.
    c. Decompose each active edge pair into a series of fragments clipped to the pixel boundaries. Calculate the z depth of each fragment.
    d. Sort the fragments for each pixel, and discard those that will not be visible in the final scene.
    e. Shade all pixel fragments, interpolating the necessary values across the scan line.
    f. Anti-alias each pixel array with itself and the background.
    g. Write the scan line to the frame buffer.

Steps 2, 3, 4, 8, 10b, 10e, and 10f can all be vectorized. Step 9, the insertion of each edge pair into its y bucket, is done only once and therefore need not be vectorized. There is one y bucket for each scan line, and each contains a list of the edge pairs (polygons) that become active on that line.

Steps 10c and 10d are difficult to vectorize because they require a large series of small lists to be sorted and culled. Once the list is sorted, polygon fragments are stacked until the entire pixel is filled, and all the fragments are discarded. This step also requires area calculation for each pixel fragment and is difficult to vectorize.

The scan-line z buffer algorithm was written entirely in C. The CONVEX vector C compiler does an excellent job of vectorizing the complex data access methods in C. The Supercomputer Graphics Group at Ohio StateUniversity programs exclusively in C, and the CONVEX vector C was and is the only existing vectorizing version of the C language. It provides an extremely high level of optimization and vectorization.

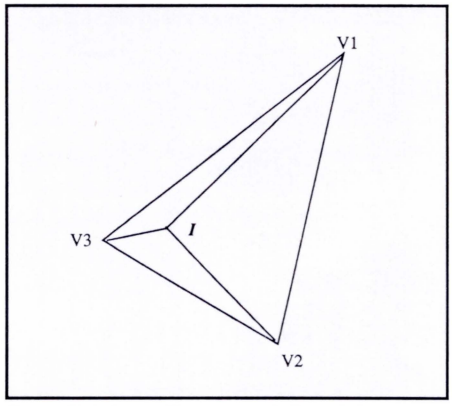


Figure 7: Intersection computation. Intersection point I is weighted toward vertex V3.

The scan-line z buffer algorithm provides an efficient vectorized rendering system for the production of high-quality computer images. It enjoys a greater than two-to-one advantage in speed over a scalar version. Surprisingly, the normally expensive shading process makes up less than ten percent of computation time on the vectorized version of the algorithm. It is the sorting steps and area calculation that comprise the bulk of the execution time, because those steps could not be vectorized.

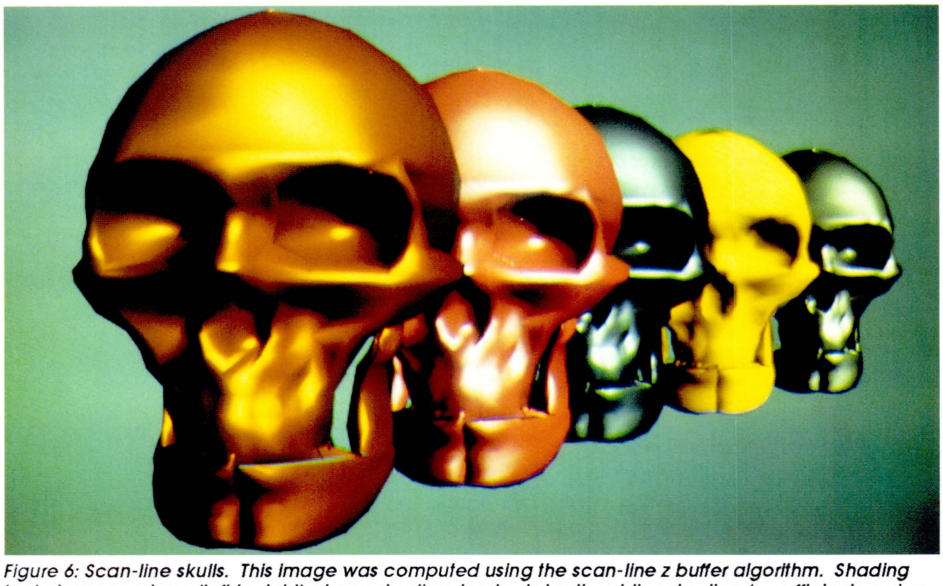


Figure 6: Scan-line skulls. This image was computed using the scan-line z buffer algorithm. Shading techniques used are (left to right) phong shading, lambert shading, blinn shading (an efficient version of phong), modified blinn, and cook-torrance (a technique that simulates metallic objects). This image required one minute of calculation on a CONVEX C1.

SAMP: A stochastic sampling rendering algorithm

The sampling algorithm samples the image plane multiple times for each pixel and averages the result to generate the image. However, a technique called *stochastic sampling* effectively eliminates the aliasing problems expected with this kind of solution.

The human visual system avoids aliasing by converting to noise the signals beyond its sampling rate. Small details are indistinguishable. This is because human visual receptors are *randomly* distributed rather than being arranged on a grid. Aliasing, a direct result of sampling on a regular grid, is therefore avoided.

To mimic this behavior in a rendering algorithm, moving the samples around randomly simulates the human visual system. This distribution, called a *Poisson disk,* converts to noise the frequencies above the sampling limit.

Moving samples around on a regular grid is called *jittering* and results in a much simpler algorithm than the scan line. Because the image plane is sampled at a single location, the hidden-surface problem is reduced to determining which polygon face is on top. All other polygons are disregarded. Polygon fragments need not be computed, and polygons are not clipped to pixel boundaries for area calculation as in the previous algorithm. The anti-aliasing process is applied uniformly across the entire sampling grid.

This algorithm can be almost entirely vectorized. As before, the transformation process and the computation of surface normals vectorize easily. However, in this case, no sorting is done and most steps vectorize directly.

1. Set the defaults for the scene and object. Read scene description and object.
2. Apply the world space transformation to the object.
3. Compute the surface normals.
4. Apply the screen space transformation to the object.
5. Compute the reflection vector and normalize the light.
6. Discard back-facing polygons, polygon-bounding-box information.
7. For each scan line:
   a. Build a list of polygons that can intersect the pixels in the current scan line.
   b. Load the background color into the scan line.
   c. Compute the intersections.
   d. Prune the tests that did not intersect.
   e. Sort the remainder by depth for each pixel, discarding all but those on top.
   f. Shade the remaining pixels.
   g. Reconstruct the remaining intersections into the current scan line.
   h. Write the scan line to the frame buffer.

Step 7a can take several forms. The goal is to produce a list of polygons within the boundaries of each pixel and may affect the pixel's color. The efficiency of this step determines the algorithm's efficiency. If no polygons are missed, the resulting image will be correct. The simplest technique is to list every polygon for every pixel. A small refinement computes the bounding box for each polygon and only includes those polygons whose bounding box contains the current pixel.

In step 7c, each pixel point is tested against the list of possible polygons, and a series of factors is computed. These factors later determine the values of the surface normal and other parameters at the intersection point. The intersection test uses a property of the cross product—for a triangle, the cross product of the edges at each vertex is the same. The cross product of a vector formed by the edge and a vector formed by the included vertex and the intersection point is also computed for each vertex. By dividing this number by the edge cross product, three factors are computed, one for each vertex. These numbers always sum to one and give the effect of each vertex on the intersection point. This computation is the heart of stochastic sampling and vectorizes completely.

Step 7d prunes the intersections that failed. These will have one or more factors that are negative. Next, invisible intersections are discarded, and what remains is a list of pixels to be shaded. This list differs from the list derived from the scan-line method. Because each intersection has had the three factors computed, finding the surface normal is merely a matter of averaging the three vertex surface normals using the factors. The same is true for color and other shading information. This makes the shader both simplistic and completely vectorized. Finally, the reconstruction process is also vectorized. This results in an algorithm that is almost entirely a vector process.

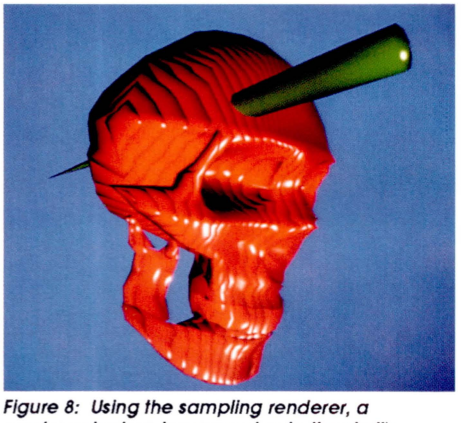


Figure 8: Using the sampling renderer, a corduroy texture is mapped onto the skull's surface normal. It was computed in about ten minutes on a CONVEX C1.

Stochastic sampling generates anti-aliased images with as little as three times oversampling. It is not as fast as scan-line, but it can be moved to new machines and architectures more quickly. Also, difficult techniques such as motion blurring and shadows are easier to implement.

Improving the polygon selection criteria of step 7a makes the algorithm even more efficient. The simple version of the algorithm used a bounding-box test to compose a list of possible intersections. Tests indicate that as the complexity of the image increases, the percentage of correct guesses decreases rapidly. By improving the selection test, you can increase the speed.

Conclusions and Future Work

Affordable vector processors such as the CONVEX offer an excellent solution to the enormous computing requirements of computer graphics. Because so many graphical tasks are repetitive and because they are so often applied across the entire image plane, vectorization is a natural operation.

Experience indicates, however, that machines such as the CONVEX have much greater impact on the quality of graphics than on the speed of production. As is so often the case, users simply increase the complexity of their work until the available resources are strained. We are used to the "ten-minute rule" in which high-quality imagery always requires ten minutes per image, regardless of the available computer power. When moving from a slow, scalar processor to the CONVEX, users simply add more detail, texture, and objects to increase the rendering time to ten minutes. Why ten minutes? Perhaps because it represents the maximum time users are willing to wait for an image to calculate. Images that require longer than ten minutes to calculate are best done in a batch mode rather than interactively. It seems likely that even with a machine an order of magnitude more powerful than a CONVEX, images will still require about ten minutes to compute.

One unfortunate side effect of vectorization is memory consumption. Because vectorization requires that data be prepared in blocks for processing, it is very easy to tend towards large tasks that include huge arrays for data storage for vectorization. Without a large physical memory, this can lead resulting loss of efficiency. The simplest solution is to purchase more memory. Because the CONVEX system can accept a Gbyte (or two in some systems) of physical memory, users are more likely to run out of swap space long before they exhaust the physical memory expansion. This is an expensive course of action, however.

The CONVEX system is unique because it provides true virtual memory and page swapping. Users on Cray® X-MP systems suffer because the Cray provides neither virtual memory nor swapping of any kind. Tasks are limited to small memory models and must use some form of internal data swapping, such as writing temporary files, to reduce their memory consumption.

Because the sampling renderer was designed to be highly machine independent, a flexible memory model was required. A solution had to provide opportunities for vectorization but also allow the program to run on extremely memory-limited hardware. The solution lay in the unit that was used for the original vectorization—the scan line. When the algorithm was first written, all vectorization was based on processing a single scan line at a time. Little notice was given to the actual size of the resulting array; the decision to vectorize on scan-line boundaries was one of convenience rather than plan. By writing the algorithm to divide each scan line into some number of pieces and to operate on those as the fundamental unit of vectorization, significant memory savings were made. In addition, the algorithm is configurable. For a large memory machine, scan lines are not divided; while for smaller machines, scan lines can be divided until memory consumption is appropriate.

The Computer Graphics Research Group at the Ohio State University received its CONVEX C1 in late summer of 1986. We have consistently written our code in C, and the CONVEX operating system is both efficient and robust. We are developing a number of new applications, including extensions to the algorithms discussed here as well as new techniques. The vector architecture of the CONVEX provides us with an excellent research platform for computer rendering.

Corporate Headquarters
CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
Phone: 214 497-4000
Fax: 214 497-4848

European Headquarters
CONVEX Computer Corporation
Randalls Research Park
Randalls Way
Leatherhead
Surrey, UK KT22 7TS
Phone: 011-44-372-386696
Fax: 011-44-372-375877

Asia/Pacific Headquarters
CONVEX Computer Corporation
1 Scotts Road
#25-06 Shaw Centre
Singapore 0922
Phone: 65-733-4355
Fax: 65-733-4354

080-001035-000
Printed in U.S.A.