Directly Rendering Non-Polygonal Objects on Graphics Hardware using Vertex and Fragment Programs

Russell M. Taylor II
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

A method of exactly rendering non-polygonal objects as primitives using vertex and fragment programs on a commodity GPU is presented. This method is demonstrated for spheres, implicit quadric surfaces, and a repeated-slat object, but it generalizes to any object for which the location and surface normal at a ray intersection can be computed.

CR Descriptors: I.3.3 (Display algorithms). I.3.5 (Curve, surface, solid, and object representations).

Additional Keywords: Implicit Surface Rendering, Ray tracing.

1. Introduction

Many objects in man-made environments consist of unions of quadratic surfaces (cylinders form gun barrels, lamp stands, telephone poles, staves, curtain rods; ellipsoids form door handles, eyes, playground balls; cone sections form cups, candleticks, table legs, lamp shades), or repeated structures (louver blinds, collenades, building facades). These objects are rendered in current real-time systems by approximating the objects with triangles, resulting in contour and lighting artifacts.

This paper demonstrates that these objects can be rendered in real time more efficiently than with any prior method. The insight that enables real-time rendering of non-planar objects lies in using the screen-space depth buffer for visibility sorting while performing viewing-ray intersection with each object (in its object space) in a fragment program.

1.1 Prior Work

Direct rendering of non-planar objects include Brooks' reprogramming of the quadratic expression evaluator on Pixel-Planes 4 to compute quadratic approximation to spheres [Fuchs95] and Kautz and Seidel's method that uses displacement mapping and renders several volumetric slices through the visual hull of an object. [Kautz01] Olano described a framework to transform and interpolate parameters for non-polygonal primitives in screen space using a Z-buffer architecture on the UNC PixelFlow machine, and also described how it could be used to drive algorithms that subdivide non-polygonal primitives. [Olano98] Shade et. al. describe depth sprites and layered-depth images that provide sampled object representations including depth and color. [Shade98]

Previous attempts to perform ray tracing on graphics hardware solve the visibility sort in object space within fragment programs. [Purcell02][Carr02][Hasselgren02]

1.2 New: Commodity Non-Polygonal Objects

The approach presented here relies on the standard Z buffer to perform visibility sorting between objects. It implements primitives that represent *non-polygonal objects* (NPOs) using polygonal stand-ins on commodity graphics hardware. Unlike [Olano98], it requires neither that the primitive be transformable into screen space nor that it be possible to interpolate its shape parameters in screen space. Unlike [Kautz01], it requires only polygons enclosing the screen-space extent of each NPO. Unlike [Shade98], it renders exact representations of spheres and other surfaces and enables the use of any invertible modeling transformation that preserves parallelism. Unlike [Purcell02] and [Carr02], it renders non-planar objects in real time, mixed with standard primitives, without read-back to the CPU.

2. Implementation

Real-time rendering of NPOs has been achieved on commodity graphics hardware by taking advantage of programmable vertex and fragment processing. The basic approach is to perform ray-object intersection in object space during fragment processing, compute the object's color and screen-space depth, and store the results.

Application: Each NPO is represented by:

- one or more polygons that cover the object's screen-space projection (called the *bounding geometry*), and
- per-object parameters sufficient to describe the object.

A sphere can be described by four parameters: the (X, Y, Z) location of its center and *radius*. A quadratic surface can be described by the eight parameters of its implicit equation: $A+BX+CY+DZ+EXY+FXZ+GYZ+HX^2+IY^2+JZ^2 = 0$. These parameters can be embedded directly in the fragment program or passed as uniform parameters.

Vertex: In the proof-of-concept implementation described here, the vertex program computes the object-space eye point and four coefficients *A, B, C, D* that can be used to determine the clip-space depth of a point in object space *(X,Y,Z)* by computing $AX + BY + CZ + D$.

The proof-of-concept implementation has the vertex program pass the linearly-interpolated object-space vertex positions for the bounding geometry coordinates in object space. As with any rendering, the vertex program also computes the homogeneous clip-space position of the vertices for the bounding geometry

Fragment: The fragment program determines the first intersection between the NPO and the ray from the eyepoint towards the interpolated vertex position. It then computes the appropriate clip-space depth value for that point using the *A,B,C,D* coefficients. If the ray does not pierce the object, the pixel is discarded (the proof-of-concept implementation sets the depth past the yon plane). This depth value can be directly used to order all NPOs in the scene. If NPOs are rendered in the same scene as polygonal objects, this depth can be converted into the 1/Z space used by the standard rendering pipeline using the hither and yon depths (passed as uniform parameters):

$$Depth = (1 / Z_{hither} - 1 / Z) / (1 / Z_{hither} - 1 / Z_{yon})$$

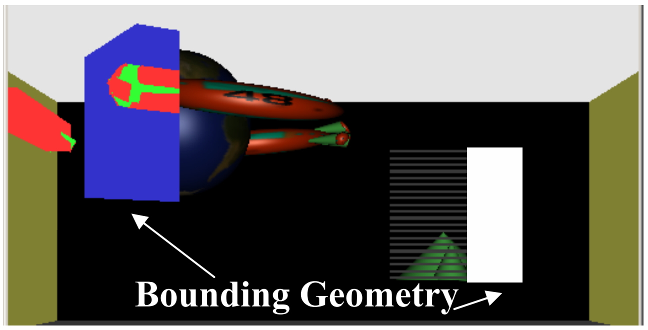
If the object is textured, the object-space texture coordinates are determined based on the point of intersection. The texture coordinate calculation is based on the object type: for a sphere, the proof-of-concept implementation uses longitude and latitude.

In the proof-of-concept implementation, the light comes from the eyepoint, enabling the fragment program to reuse the eye-to-vertex vector as the lighting direction. The object-space light position could be passed as a uniform parameter and used to light the object.

Discussion: The calculations described above can often be shifted to a different processor (application, vertex, fragment) to alleviate system bottlenecks. For example, the depth *A,B,C,D* coefficients could be computed by application code and passed to the fragment program as uniform parameters for each object based on the current modeling, view, and projection transformations. They can be performed using different algorithms (solving for *A,B,C,D* could be done using different points or different approaches). Figure 1 shows a general quadratic implicit-function NPO rendering conical mountains out the window.

The presented technique can be used on a broader set of objects than implicit equations. For example, figure 1 shows a *louver blind* NPO. The fragment shader, translates the bounding-geometry pierce point into the range $-1 \leq Y < 1$ and then intersects the view ray starting there with the "slats" at Y = -1 and Y = 1. This intersection is then translated back to the appropriate location. The blinds are opened and closed by shearing and anisotropically scaling the bounding geometry. In this manner, an entire set of blinds can be rendered using one bounding box.

The ray-object intersection is performed in object space because modeling transformations can change the form of the parameters (a sphere is no longer a sphere) and because texture coordinates, if used, are required in object space.

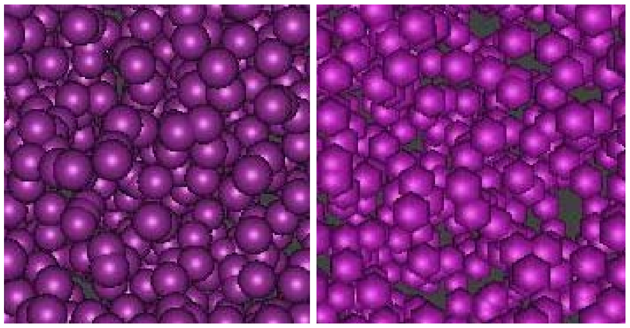


Modeling transforms can also be used to distort the shape of any NPO. The figure below shows that the modeling transform can be used to form all ellipsoids from the unit sphere centered at the origin. This works with all NPOs.

If the fragment shader computes the screen-space normal for the NPO, all commonly-used advanced rendering features such as bump maps, displacement maps, and so on can be applied. Techniques like supersampled antialiasing and shadows can be applied directly because the NPO depth rendering behaves just like triangle rendering with respect to interpolation and the depth buffer.

The presented techniques are applicable to any function that uses graphics hardware, not just rendering. Collision detection, robot motion planning, and computing distance fields are example applications that may benefit from the direct rendering of NPOs.

3. Performance



A proof-of-concept implementation was tested on a 3-year-old Pentium-III 667 MHz, with a 133MHz memory bus, an NVidia DeForce FX 5700 Ultra, AGP2, and 128MB memory rendering to a 32-bit display. A 2000-sphere scene using tight-fit cube bounding geometry rendered 37,500 colored, diffuse + specular lit spheres/second into a 700x700 window (zoom-in shown in image to the left). The comparable polygonal scene using 6x6-tesselated *glut-SolidSphere(1.0, 6, 6)* primitives in display lists ran at 32,200 spheres/second at much lower image quality (zoom-in from a different viewpoint shown in image to the right).

It is sufficient to use the screen-space-interpolated depth values for the bounding geometry as the depth value for the NPO itself if there are no intersections between NPO geometry and geometry of other objects in the scene. This reduces both parameter-passing bandwidth and the number of calculations in the vertex and fragment programs, increasing the performance beyond that stated above.

4. Limitations

The bounding geometry for NPOs do not by themselves clip the object geometry: if the viewer looks through the side face of an axis-aligned bounding box around an NPO like a cylinder, they will be able to see geometry that extrudes below the bottom face. If this is undesirable, the NPO fragment program must explicitly check for intersections lying outside the bounding geometry (this is done for the cones shown above). This is not required for NPOs whose implicit functions have zeroes only within the bounding geometry used to select where on the screen they will be drawn (spheres).

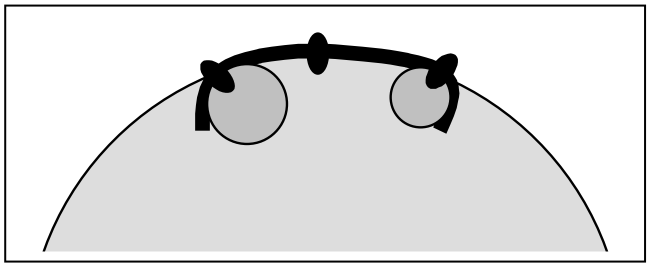
5. Availability

Source code for the programs used to produce all images in this will be made available upon publication, and will be made available to reviewers upong request. A provisional patent application has been filed on this technique by the University of North Carolina, which we intend to let expire in Spring 2004.

6. Future Work

Although spheres and other quadric surfaces together with specific implementations of primitives for repetitive and nested surfaces provide a set of interesting objects that can be rendered, many surfaces in use today possess more complex curvature than can be captured by the NPOs described above. The curved shapes are often represented (and designed) using spline surfaces, which are then approximated by a tessellation of triangular facets for rendering.

An extension of the above techniques enables us to use quadratic or higher-order surface approximations in place of the first-order triangular tessellation. Rather than planar geometric surfaces with interpolated normals (as Phong described), we can have higher-order surfaces with interpolated parametric descriptions. A simple example is that of a convex patch with varying curvature, a cross-section of which is shown in the figure below.



At each location, the patch can be approximated by the osculating circle (ellipsoid in 3D) that matches its position and curvature at that location. A sampled representation of the surface could then be provided by a set of such circles as shown in the figure below. By interpolating the parameters (center and radius) between points, we can form an infinite set of circles that approximate the surface. Unlike triangular tessellation, this approximation has smooth silhouettes (C1 continuous) and changes in curvature (C0 continuous).

We can implement such an approximation in 3D using the techniques presented earlier using a polygonal bounding geometry for the ellipsoidal approximation surface and associating the circle parameters with each vertex and then interpolating them between vertices. We can also embed the parameters into a texture and then map the texture onto the bounding geometry, providing an even finer sampling for the ellipsoidal approximation surface. As each view ray pierces the bounding geometry, interpolation is used to determine the parameters for the primitive approximating the surface at that location and the ray is intersected with this primitive.

This generalizes beyond convex surfaces and ellipsoidal approximations through the use of quadric or higher-order approximations. Each ray is still compared against only one NPO, but the parameters for that NPO are determined by interpolation (possibly after texture look-up) rather than being the same across the whole bounding geometry.

Acknowledgements

Thank you to David Borland, Mark Harris, and Alvin Richardson for many discussions about programmable graphics hardware. The Cg developer tools and example programs made it possible to implement the code described here in under a week.

REFERENCES

Carr, N., J.D. Hall, J.C. Hart, "The Ray Engine," *Graphics Hardware 2002*, pp. 1-10.

Fuchs, H., J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F.P. Brooks, J. Eyles, J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *SIGGRAPH '85*, pp. 111-120.

Hasselgren, J., "Fragment Program Raytracer," Cg shader demo program, http://www.cgshaders.org/shaders/show.php?id=36

Kautz, J. and H.P. Seidel, "Hardware Accelerated Displacement Mapping for Image Based Rendering," *Graphics Interface 2001*, pp. 61-70

Olano, M., A. Lastra, "A Shading Language on Graphics Hardware: The PixelFlow Shading System," *SIGGRAPH '98*, pp. 159-168.

Purcell, T.J., I. Buck, W. Mark, P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *SIGGRAPH 2002*, pp. 703-712.

Shade, J.W., S. Gortler, L. He, R. Szelinski, "Layered Depth Images," *SIGGRAPH '98*, pp. 231-242.