3-D Graphics in Physics

Jeffrey Naset and C. J. Stanton

University of Florida, Gainesville, FL

An overview of methods available to physicists for displaying and manipulating three-dimensional graphics, with a detailed description of the OpenGL API as it pertains to typical physics problems.

3 August 1999

It is no surprise that three-dimensional graphics are useful to a physicist. The world in which we live is three-dimensional, and thus the laws of physics are inherently three-dimensional also. We would like to present depictions of physical fields, such as density, charge, or quantum-mechanical wavefunctions, which are defined in three-dimensional space in an intuitive way to help our readers and viewers easily understand important qualitative and quantitative features about them. There are some minor obstacles; namely, most of the media in use today and for the foreseeable future are two-dimensional, paper, transparency, and video being most prevalent. With a little effort, however, and the right tools, one can make great strides in elucidating various phenomena for one's readership.

There are a number of useful tools available today which are useful in the production of three-dimenstional graphics. Visualization applications such as IBM's Data Explorer allow one to manipulate sets of data interactively, and quickly generate high-quality graphics. Cross-platform APIs such as OpenGL enable one to write portable source code to generate graphics in a reasonable amount of time. File formats such as VRML (Virtual Reality Modeling Language) allow digital publication of 3-D graphics such that end users can pan, zoom, rotate, and fly through data sets.

The choice of tool will depend on the intended usage of the graphics, and the amount of time available to generate them. For rapid development and high quality rendering a full-featured application solution like Data Explorer (now OpenDX) may be the best choice, as it is able to import data sets in many formats and features a graphical programming language. Data Explorer's strengths include tested and debugged high-

quality rendering algorithms, its easy-to-use programming language, and its adaptability to most pre-existing data files. Its weaknesses are non-portability (in the sense that a DX program requires the DX package and the original data sets to run) and speed (DX is not suitable for computation, so data must be imported in the form of huge files which are slow to process). Other applications, such as Mathematica and Maple, are more suited for computation, but include some 3-D graphics support. Their output may be suitable for some simple applications (for instance, when one coordinate is a single-valued function of the other two), but in general they are not good mechanisms to view arbitrary data.

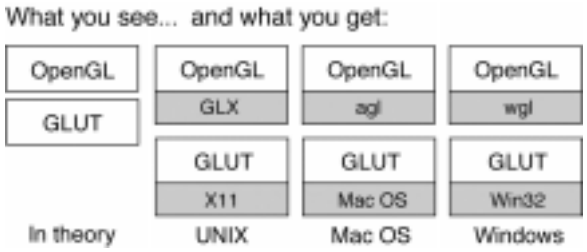
The most general solution is to write one's own application using available APIs and libraries. This can involve duplicating a significant portion of user interface and system-specific code in some cases, but for the most part the libraries and templates allow one to focus on the data and presentation. Advantages of this approach include the ability to distribute an interactive program in source or binary form to enable users to explore the data at will; efficiency, because a particularly efficient proprietary data storage can be used if desired, or the data might even be computable on the fly; and flexibility, since you have the full power of a programming language like C and typically the interfaces of UNIX and X. For the rest of the paper I will be talking about how to write a program in C using the OpenGL API and the GL Utility Toolkit (GLUT). A suitably written OpenGL/GLUT program will compile on most popular personal and time-sharing computer operating systems without modification or specialization. Also, because C is a compiled language, the results will typically be faster than the interpreted programming languages used by the monolithic applications mentioned previously.

Overview of the OpenGL API

OpenGL is designed to be cross-platform. It includes no reference to any operating-system-specific entities of any kind. This means that OpenGL has no notion of a window, or a computer screen, or even an offscreen buffer. The only way to acquire an OpenGL context (the context encapsulates all of the state associated with OpenGL) is via an operating-system-specific API. Fortunately, the freely-available GLUT library

provides a universal way to acquire an OpenGL context in a user program. It maps generic elements such as windows, menus, and mouse buttons to the most appropriate facility on the host operating system. Unfortunately GLUT cannot provide a comprehensive user interface experience (it has no analogue to X widgets or Mac OS/Windows controls), necessitating a small amount of platform-specific code in some cases. GLUT will transparently use an appropriate protocol to communciate with OpenGL on your platform: GLX/X11, agl, or wgl.

OpenGL presumes very little about the host operating system or hardware. Graphics commands may be executed by hardware or software. The frame buffer may be true- or indexed-color. On all platforms, however, the mathematical model will be the same. You will execute calls to the API to generate primitives (points, lines, triangles, and quadrilaterals) in 3-D space. OpenGL will transform the vertex coordinates and certain other properties (normal coordinates, for example) you give it according to user-defined *modelview* and *projection* matrices. If lighting is enabled, OpenGL will perform the lighting calculations at each vertex; otherwise you specify the color of each vertex yourself. In either case, the pixels in the resulting image will be colored by interpolating the colors from the vertices of the primitive. (By default—you can specify flat shading if you prefer) Coordinates in OpenGL consist of four floating-point coordinates: *x*, *y*, *z*, and *w*. The *w* coordinate is used to facilitate perspective transformations and translation matrices. Before OpenGL transforms 3-D coordinates to 2-D coordinates for rendering, the *x*, *y*, and *z* coordinates are divided by the *w* coordinate.



Hidden surface removal is performed in OpenGL by specifying *face culling*, *Z buffering*, or both. Face culling relies on the fact that the vertices of a forward-facing triangle or quadrilateral must be specified in counter-clockwise order with respect to the viewer in order for lighting to work properly (the viewer's position is implied by the value of the projection matrix). If a primitive has clockwise vertices after transformations are applied when face culling is enabled, the primitive is discarded. Z

buffering extends the frame buffer with an additional plane of pixels of 16- or 32- bit depth. Whenever a pixel is to be stored in the frame buffer, the transformed *z* coordinate is compared to the existing value in the Z buffer, and, if it is greater (indicating that the new pixel is being generated by a primitive further away from the viewer than the primitive which resulted in pixel being drawn previously) the new pixel is discarded. Note that while face culling operates on primitives, Z buffering operates on pixels. Z buffering is almost always necessary except if a more intensive operation called *Z sorting* is used. (Z sorting is required to properly implement exotic effects such as partial transparency) Regardless of whether Z sorting or Z buffering is used, face culling can reduce rendering time by up to half and should be used whenever back-facing primitives can be safely discarded (such as when it is known that all primitives together form solids with unbroken faces).

The hardest part of working with OpenGL is manufacturing complex objects. Since OpenGL only allows you to draw points, lines, triangles, and quads, you must build up more complicated objects from those primitives. (Typically triangles work best since their vertices are by definition coplanar. Quadrilaterals may not be and are arbitrarily split by OpenGL in that case.) There is an auxiliary library which usually ships with OpenGL called GLU (GL Utility) which provides spheres, cylinders, and the teapot by generating triangles for you.

The Project

Source code for this project is available at: http://www.naset.net/~jeff/reu99/

The goal of the REU project was to write a program capable of rendering an isosurface of some scalar function. The program was to be able to render the results interactively in a window or to produce VRML output for storage or transmission over the web. Since by its nature the project was concerned with implementation details, I will describe some of the issues I encountered while developing the program and show the solutions.

The first step in plotting an isosurface is to find a mesh of triangles closely approximating the true isosurface. It is convenient to partition space into cubes or

tetrahedra and check for intersections through each such cell. I chose to use cubes since their geometry is convenient in Cartesian coordinates. The algorithm is as follows:

1. Divide the region to be graphed into coarse cubes
2. For each cube, check each vertex to see if it lies inside or outside the surface
3. If all vertices are inside or all are outside, discard cube
4. For each edge which has one vertex inside and one outside the surface compute a point of intersection with the surface by interpolating and refining
5. Join the points of intersection to form a polygon
6. Split the polygon into triangles
7. Verify the triangles obey a counterclockwise vertex ordering; otherwise permute the vertices

The algorithm produces a series of triangles suitable for display using OpenGL or storage as VRML. Decomposing the surface into triangles is the most portable thing to do – although OpenGL supports quadrilateral primitives and VRML supports arbitrary polygons, they both end up decomposing them into triangles anyway. Having the data in a uniform format allows the OpenGL driver to render using the most specialized, efficient code path.

Because the fermi surface lives in the Brillouin Zone, it is necessary to be able to clip output to this region, and desirable to produce a wireframe representation of the region as well. (Note that OpenGL provides for six clipping planes, however this is insufficient, and not even applicable for VRML) For this reason, I have implemented primitive and geometry clipping functions. The primitive clipping function will clip triangle data to the intersection of regions described as being on one side of some plane. The geometry clipping will find a set of lines which follow the boundary of the region. Both functions are rather straightforward applications of linear algebra, so I won't go into detail about them here.

For those readers interested in reading or using part of the source code, I will now describe some details of it.

The `mesh_gen_t` structure contains all of the parameters used to generate a mesh. Most of the fields are self-explanatory. The function of which an isosurface is to be generated, however, is specified in three pieces: radial, angular, and normalization. This is done so that a complicated radial function could be cached. The normalization

function allows plotting things other than a plain isosurface by manipulating the functional values after they are computed by the radial and angular parts. Just keep in mind that what you get is an isosurface of the following function:

$$F = \text{normalization}(\ \text{radial}(r) \times \text{angular}(x,y,z,r)\ )$$

The other thing is that the `disp_func` and `disp_close` fields specify which function is responsible for storing the triangles – it could be the VRML output function, the OpenGL output function, the floating-point buffer function, or some other thing. These two fields must be initialized by calling the appropriate output initialization function, for example `init_gl_disposer` or `init_float_buf`. The initialization functions set up other private fields and prepare the output function to handle data.

Be sure to check out the web site about this project to see it in action and download the source and examples.