

Developing 3-D Analyst Enhancements Using OpenGL

Next Generation Command and Control System (NGCCS)
Tactical Operations Center (TOC) 3-D Program
Concurrent Technologies Corporation (CTC)
Jeremiah Montgomery, montgomj@ctc.com, Software Engineer
Christopher Moore, moorec@ctc.com, Software Engineer

June 21, 2004

***Abstract:** ArcGIS provides users with powerful tools to create advanced 3-D mapping applications. These tools include the ArcObjects Application Program Interfaces (APIs), as well as a complete set of ActiveX controls. Notwithstanding all of its power and versatility, ArcGIS still suffers from some limitations with regard to the rendering of complex 3-D graphics. Fortunately, the ArcObjects 3-D Analyst architecture provides a means to address these limitations by providing an "entry point" in its 3-D controls into which developers can plug custom, OpenGL-based graphic rendering components. This allows for easy integration of high-speed, OpenGL components into existing ArcGIS architecture components.*

The approach outlined above has been taken with prodigious success by CTC in development of the TOC 3-D Program for the U.S. Army Project Manager, Ground Combat Command and Control (PM GCC2). The TOC 3-D Program is an early adopter and integrator of the ArcGIS 9.0-based Commercial/Joint Mapping Toolkit (C/JMTK) within its developed software products.

Introduction

With the advent of its 9.0 release, the ArcGIS 3-D Analyst extension has introduced greatly improved 3-D visualization capabilities. While these 3-D visualization capabilities are significant and extensive, there are times when the developer may need to extend the rendering provided by the 3-D Analyst. Examples of such extensions include sensor cones, track lines, as well as many other highly specialized graphic objects that are not necessarily supported by the 3-D Analyst solution "out of the box."

Fortunately, ArcGIS products are built to be extended, and 3-D Analyst is no exception. ArcGIS provides mechanisms to extend the object model, User Interface, display, and also provides many other ways to seamlessly integrate functionality into the ArcGIS environment.

In particular, 3-D Analyst provides the developer with the means to extend the rendering capabilities of its products and to perform highly customized rendering. One such method to enhance and extend 3-D Analyst is by using custom OpenGL rendering.

This paper explores the mechanics of integrating OpenGL-based custom rendering solutions into 3-D Analyst. This paper first provides useful background information on 3-D Analyst, OpenGL, and other topics. Next, the paper explores various aspects of OpenGL development in further detail. Then, methods for integrating OpenGL and 3-D Analyst are shown. Finally, these techniques are shown together and demonstrated using several sample and real world applications.

This paper explores OpenGL capabilities primarily in the context of the premier 3-D Analyst product, ArcGlobe. Therefore, while 3-D Analyst includes several different products, for the purposes of this paper, 3-D Analyst refers primarily to the capabilities provided by ArcGlobe. In this paper, the terms 3-D Analyst and ArcGlobe are often used interchangeably. However, it is important to note that the approach outlined will also work with other 3-D Analyst products, such as ArcScene. Additionally, this paper assumes some comfort and familiarity with the following concepts: ArcGIS 3-D Analyst, 3-D graphics/visualization, and general computer programming.

Background

This portion of the paper presents some basic background information on 3-D Analyst/extensions, C/JMTK, TOC 3-D, and OpenGL.

About 3-D Analyst and ArcGlobe

ArcGlobe represents a paradigm shift in 3-D Analyst from isometric to global 3-D perspective viewing. The former technique is embodied in the ArcScene application, which presents to the user a “postage stamp” view of a very small portion of the earth. Within this view there is little to no sense of global position or geo-curvature—hence the “isometric” moniker. Alternately, the latter technique, global perspective viewing, is embodied in the newly released ArcGlobe application. ArcGlobe presents the user with a “whole earth” picture, fitting each particular user’s area(s) of interest visually into the context and proportion of the full earth itself. As such, ArcGlobe represents a new innovation in Geographic Information Systems (GIS) visualization—an opening of the door toward a future of ever-increasing and *truly geospatial* awareness. For more information on 3-D Analyst and ArcGlobe, consult www.esri.com/software/arcgis/extensions/3danalyst or www.esri.com/library/fliers/pdfs/arcglobe.pdf.

About ArcGIS 3-D Analyst Extensions

When developing ArcGlobe-based applications, two options present themselves to the software architect. This is because ArcGlobe is made available by ESRI in the following forms:

- **ArcGlobe Application**
 - The ArcGlobe application is a fully-functional whole-earth GIS data viewer with many built-in data manipulation and viewing options. It ships with the 3-D Analyst extension to the ArcGIS Desktop suite of end-user products.
- **ArcGlobe Globe Control**
 - The Globe control is an ActiveX control available to application developers using Visual Basic, C/C++, or a .NET programming language (such as C#). It ships with the 3-D Analyst extension to the ArcGIS Engine Software Development Kit (SDK).

Given these two available methods for extending ArcGlobe, the developer must choose whether to *extend* the existing ArcGlobe application by writing a “plug in” to the desktop application’s user interface, or whether to *embed* the Globe control into a custom, standalone application.

As shown in Figures 1a and 1b, the functionality and main display of the ArcGlobe application and the Globe control are very similar.

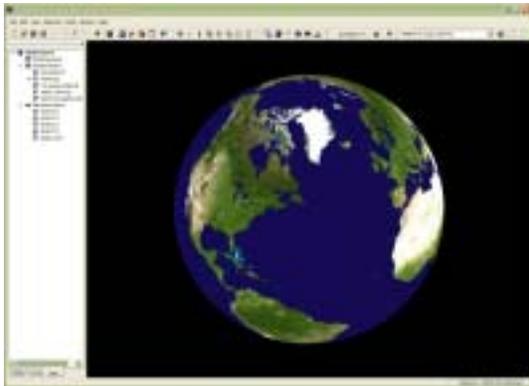


Figure 1a.
ArcGlobe Application

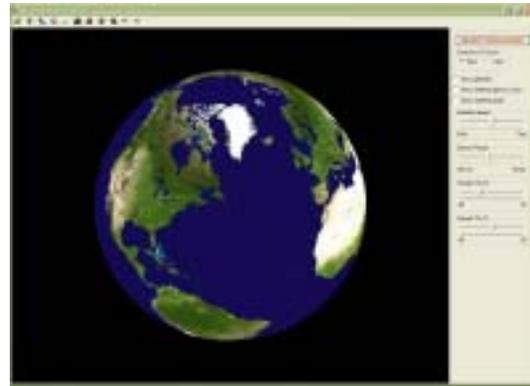


Figure 1b.
Sample Application (Globe Control)

As can be seen, the visual differences between the two forms of ArcGlobe are negligible; therefore, it is primarily up to the system developer to consider factors, such as required functionality, license costs, development and end-user complexity, and others factors when selecting a target display. The techniques documented in this paper work with either method of employing and extending 3-D Analyst.

About the C/JMTK

The C/JMTK is the next generation geospatial visualization tool for the Department of Defense (DoD) Common Operating Environment (COE). C/JMTK 1.0 is based on the 9.0 pre-release of ESRI’s ArcGIS suite of products. It is a comprehensive toolkit of ArcGIS software components for the management, analysis, and visualization of map and map-related information in both 2-D and 3-D. As such, the C/JMTK includes ArcGIS

Engine, ArcIMS, ArcSDE, 3-D Analyst, as well as several other ArcGIS extensions. For additional information on C/JMTK objectives and capabilities, consult www.cjmtk.com.

About TOC 3-D

The TOC 3-D effort is being developed by *CTC* under the NGCCS TOC 3-D program. TOC 3-D is an initiative by the PM GCC2 under the auspices of the Program Executive Office, Command, Control and Communications Tactical (PEO C3T).

The TOC 3-D initiative integrates advanced GIS technologies, 3-D Visualization, information management solutions, evolving communications architectures, and Internet technologies to meet the Army's requirements to display battlespace-relevant data including maps, intelligence information, assets, and live feed data in a space that incorporates position, orientation, and time. This provides the capability to overlay data from many battlespace-relevant entities. The TOC 3-D system interprets the data received and displays the following information:

- GIS data.
- Geo-referenced MIL-STD-2525B symbology.
- Topographic analysis data.
- Integrated Common Tactical Picture (CTP)/Common Operational Picture (COP).
- Situational Awareness.

Integrating and validating the visualization capabilities of C/JMTK are major objectives of the TOC 3-D program. Using the C/JMTK, TOC 3-D has successfully prototyped the capability for the U.S. Army to add full-dimensional viewing of the battlefield. This full-dimensional viewing capability includes 2-D views, 3-D perspective views, and 2-D/3-D views with a temporal component.

In summary, the TOC 3-D Application combines all relevant data into one application that can be used by combatant commanders to make better-informed decisions. For more information on the TOC 3-D Program, contact Tim Barnes, Program Manager, *CTC*, at barnest@ctc.com.

About OpenGL

OpenGL is a platform-independent API that allows developers to directly access the Graphics Processing Unit (GPU) or other video acceleration hardware resident on the development and deployment platforms. OpenGL originated in the early 1990's on high-end graphics workstations as a solely C/C++ API, but has, in the intervening years, become available on nearly every mainstream hardware and software platform and is callable from a wide variety of programming languages. For more general information and the latest news on OpenGL, consult www.opengl.org.

OpenGL Primer

Before discussing integration of OpenGL-based custom rendering solutions into the ArcGlobe control of the 3-D Analyst component, it is first useful to go over the basics of OpenGL. While a complete and thorough coverage of OpenGL would require a textbook-length discussion, a basic introduction is presented. This portion of the paper provides a programmer's introduction to the basics of OpenGL needed to understand a sample application. This discussion assumes that the reader has some basic knowledge of linear algebra and programming.

Rendering Process

The process for initializing and using OpenGL to render graphics in a window can be summarized in the following process:

- **Initialization.**
 - Done once, at startup:
 - Create the OpenGL rendering context.
 - Set the viewport.
 - Set the perspective.
- **Rendering.**
 - Done once per frame:
 - Save the current state of any settings to be modified.
 - Modify the state of appropriate settings.
 - Use rendering primitives to build the scene.
 - Restore the prior state of any modified settings.
 - Finalize the scene by performing a buffer swap.

Probably the most difficult and least-portable aspect of using OpenGL is the initialization of the rendering window. As can be seen from the list above, the initialization process involves creating an OpenGL rendering context (a handle to the OpenGL state machine), setting the viewport (the user's window into the OpenGL world), and setting the perspective. Setting the perspective involves telling OpenGL to render the scene orthographically (as in CAD drawings), or with true distance perspective (the default). Because 3-D Analyst handles most of this initialization, this paper does not provide a detailed description of OpenGL initialization.

As seen from the second major bullet above, the actual rendering using OpenGL consists primarily of modifying OpenGL settings and then using primitives to construct the overall scene. The following sections discuss this process in greater detail.

The State Machine

OpenGL operates its settings modification apparatus using the concept of a state machine—a single set of functions and properties, where changes made to the settings persist until they are changed again. For example, if one method in user code changes the

rendering color from black to red but does not change it back again, all rendering done in any method from that point hence will be done in red. In procedural programming terms, OpenGL settings and operations can be thought of a set of global variables and the functions that use them. Conversely, in object-oriented programming terms, OpenGL settings and operations can be thought of as a set of static fields and methods.

This notion of a state machine is important to keep in mind, as failure to use OpenGL as such can precipitate very undesirable side-effects. In fact, it is very seldom indeed that such permanent alterations are desired; more often, users only want to change settings temporarily, do their rendering, and then change the settings back. Fortunately, OpenGL provides just such a mechanism.

Attribute and Matrix Stacks

In order to facilitate a user's ability to easily store and restore current settings before and after making custom changes and performing custom rendering, the OpenGL state machine incorporates several stacks for the archival and retrieval of OpenGL attributes and matrices. For those to whom the notion of a stack is unfamiliar, consider the example of a box into which one places books. The books can be placed in the box in any order, but since each book goes on the top of the pile, the last (most recent) book placed *in* the box will be the first book taken out of the box. Appropriately enough, this organization is referred to as a Last-In-First-Out (LIFO) structure.

An attribute in OpenGL is a single property or setting of the state machine. An example of an attribute is, as already mentioned, the current rendering color. Other attributes include the style for rendered lines (solid, dashed, etc.). Attributes are also used by OpenGL as flags to indicate whether certain modes are enabled or disabled—for example, attributes are used to flag whether such modes as anti-aliasing (automatic line smoothing) or (transparency) are enabled. Any attempt by a user to utilize a feature that is currently disabled in the state machine will fail. This paper concerns itself primarily with non-flag attributes.

To allow for an attribute to be saved on the attribute stack, OpenGL provides the `glPushAttrib(...)` function, where the attribute to be saved is identified by the function's parameter. Restoration of an attribute from the attribute stack is accomplished via a call to `glPopAttrib()` function. The functions used to modify an attribute depend on the attribute being modified. For example, `glColor3d(...)` is used to change the rendering color, while `glLineStipple(...)` is used to change the line style.

Matrices in OpenGL are used to track complex settings whose representations require a non-trivial representation. The most important example where a matrix is used is for the storage of the location, rotation, and scale of the three dimensional scene being viewed. In OpenGL this matrix is called the modelview matrix. Just as with attributes, changes made in one location to the contents of the modelview matrix are persisted by the OpenGL state machine until they are changed back or reset.

To allow for easy saving and restoration of matrices (and for the modelview matrix in particular), OpenGL provides the `glPushMatrix()` and `glPopMatrix()` functions. The specifics of using these functions can be seen in the source code accompanying the sample application.

Rendering Primitives

In OpenGL, primitives are the building blocks used to construct complex, high-detail scenes. These primitives are geometric entities—points, lines, polygons, cylinders, spheres, etc. By careful combination and ordering of these relatively few distinct entities, impressive conglomerate entities can be produced. A good example applicable to the current context is the base globe in ArcGlobe itself, which, when rendered using OpenGL, is comprised of a lattice of sphere and polygons:



Figure 2a.
ArcGlobe (Rendering Grid Hidden)

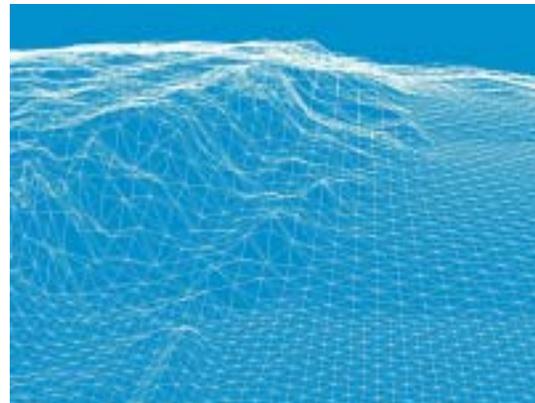


Figure 2b.
ArcGlobe (Rendering Grid Exposed)

The process used to render a primitive in OpenGL is surprisingly simple, and can be summarized by a familiar acronym: WWW. In the context of this paper, the components of this acronym stand for the steps of a simple three step primitive rendering process: *What, Where, and When*.

What

The first step in rendering a primitive using OpenGL is to declare what species of primitive is to be rendered—a point, line, polygon, etc. In most cases, this is done by calling the `glBegin(...)` function and passing as its parameter an appropriate identifier signifying the type of shape desired.

Where

The second step in rendering a primitive using OpenGL is to specify where the shape is to be drawn. All primitives are drawn using one or more points (called *vertices*). Specifying where a primitive is to be drawn, then, is a simple matter of specifying the

location of the primitive's vertices. In most cases, this is done by calling either the `glVertex3d(...)` or `glVertex3i(...)` functions one or more times (depending on the primitive being rendered) and passing as its parameters the x, y, and z coordinates of the vertex.

Notes on Shape Placement

It is worth noting at this point that shape placement in OpenGL can be accomplished in several different ways. The simplest method is described in the preceding paragraph. That is to use only the vertex specifying functions to position each vertex relative to an immutable origin. However, an alternative way to position shapes in OpenGL is to move the origin itself prior to beginning to render any primitive shapes. A shape's points are then specified as centered around the OpenGL origin, rather than at an absolute location.

This latter method, when properly used, allows for a shape to be easily moved to multiple locations without requiring a recalculation of its vertices' placements. However, its usage requires skillful manipulation of the modelview matrix—and so introduces a new level of complexity to the OpenGL development. Modelview matrix manipulations are handled via calls to functions such as `glTranslate3d(...)` and `glRotate3d(...)`.

When

The third and final step in rendering a primitive using OpenGL is to signal when the vertex specification is complete – i.e., to let the GPU know that all the points have been provided and that the primitive shape is now complete. In most cases, this is done by calling the `glEnd()` function, which takes no parameters.

Exceptions to the WWW Process

Like any general purpose procedure, the WWW process previously explicated has several partial exceptions. These exceptions pertain primarily to the rendering of non-trivial “primitives,” such as cylinders, disks, and spheres. Such objects are not rendered using the WWW process, but rather via special OpenGL objects called *quadrics*. Quadrics are part of the OpenGL Utility library (GLU), and are able to render complex shapes via single method calls—for example, `gluCylinder(...)` or `gluSphere(...)`. The reason these exceptions are referred to as “partial” is that, despite the fact that OpenGL allows for cylinders, spheres, etc., to be used as primitives, they are in fact very complex objects and are not classified as primitives.

As stated previously, the specifics of using all of the functions discussed in the foregoing sections are demonstrated in the source code accompanying the sample application.

The SMURF Method for OpenGL Rendering

All of the aspects of OpenGL programming previously discussed may be incorporated into a single, general procedure for effective rendering of each frame using OpenGL.

This procedure will be hereafter referred to as the SMURF method. It was briefly touched upon earlier. Now that a sufficient OpenGL epistemological framework has been laid out, it can be fully explained as follows:

- **S**ave the current state of any settings to be modified.
 - All attributes and matrices that will be modified in forthcoming steps must be pushed onto the appropriate OpenGL stacks using `glPushAttrib(...)` and `glPushMatrix()`.
- **M**odify the state of appropriate settings.
 - All necessary changes to attributes and the modelview matrix can be safely done at this point via functions such as `glColor3d(...)`, `glLineStipple(...)`, `glTranslate3d(...)`, `glRotate3d(...)`, etc.
- **U**se rendering primitives to build the scene.
 - Once the state machine (attributes and the modelview matrix) is appropriately configured, one or more primitives can be rendered via calls to `glBegin(...)`, `glVertex3d(...)`, and `glEnd()`.
- **R**estore the prior state of any modified settings.
 - Having completed rendering of all necessarily primitives, all changes to the state machine must be reversed by calling `glPopAttrib()` and `glPopMatrix()` in an order exactly reverse to the order in which `glPushAttrib(...)` and `glPushMatrix()` were called originally to save them.
- **F**inalize the scene by performing a buffer swap.
 - In a double-buffered OpenGL configuration, this is accomplished via a single call to `glSwapBuffer()`. In the rare case where only a single buffer is being used, no call to `glSwapBuffer()` is required.
 - Note that there has been no discussion of `glSwapBuffer()` in any of the preceding sections. The reason for this will become apparent in the next section.

The goal of the SMURF method is to exploit the full potential of OpenGL, while at the same time leaving the OpenGL context in exactly the same state as it was prior to use. Visitors to national forests and parks will often see signs posted at trailheads which read, “Take only memories; leave only footprints.” The OpenGL developer must do likewise.

Using OpenGL in ArcGlobe

Having now covered the technical approaches available when developing ArcGlobe-based applications along with the basic concepts involved in using OpenGL, this section discusses the integration of OpenGL into the ArcGlobe architecture. As soon to be demonstrated, this integration is actually quite simple to achieve.

Initialization

Recall from an earlier section that before any OpenGL rendering can be performed in a window, the following initialization steps must be completed:

- Create the OpenGL rendering context.
- Set the viewport.
- Set the perspective.

The good news for ArcGlobe/OpenGL developers here is that they need not worry about performing any of these steps. Inasmuch as ArcGlobe uses OpenGL internally for its own rendering of the base earth, it handles these initialization steps automatically.

That said, there still remains the question of how the developer is to gain access to ArcGlobe's internal OpenGL process, in order that custom rendering code may be included. This question is answered in the next section.

Creating an OpenGL Portal in ArcGlobe

In order to gain a foothold in ArcGlobe's internal rendering process, the developer must create an event handler for the globe display's outbound `AfterDraw` event. When using the ArcGlobe Globe control in an embedded application, the relevant C# code to create such a handler looks as follows:

```
using ESRI.ArcGIS.Analyst3D;
using ESRI.ArcGIS.GlobeCore;

...

private IGlobeDisplayEvents_Event globeDisplayEvents = null;

...

protected void CreateOpenGLPortal()
{
    globeDisplayEvents =
        axGlobeControl1.GlobeViewer.GlobeDisplay as IGlobeDisplayEvents_Event;
    if(globeDisplayEvents != null)
    {
        globeDisplayEvents.AfterDraw +=
            new IGlobeDisplayEvents_AfterDrawEventHandler(OpenGLRender);
    }
}

...

protected void OpenGLRender(ISceneViewer viewer)
{
    // perform custom OpenGL rendering here
}
```

There are several things to note in the sample code. The first is the inclusion (via `using` directives) of both the 3-D Analyst and ArcGlobe core class libraries. This step is more a matter of convenience than sheer necessity, as it simply saves the developer from having to fully qualify (using the `ESRI.ArcGIS.` prefix) every 3-D Analyst or ArcGlobe interface used in the code.

The second point to note with regard to the sample code is that the `IGlobeDisplayEvents_Event` reference used to create the `AfterDraw` event handler is persisted throughout the lifetime of the code via a private class field. This is required in order to assure that the event handler is not “forgotten” by the event generator, which would cause it to cease receiving `AfterDraw` event notifications.

The last important point to note is not the event handler creation that takes place in the `CreateOpenGLPortal` method (which simply follows the standard method for creating event handlers in C#), but rather the signature of the event handler (`OpenGLRender(...)`) itself. This method, though it can be of any protection level (public, protected, internal, etc.), *must* return `void` and take a single `ISceneViewer` parameter. Only a method with this exact signature can be used as the `AfterDraw` event handler for OpenGL rendering in ArcGlobe.

Once the `AfterDraw` event handler is created, OpenGL function calls may be made from anywhere within its method body. There is never a need for the event handler to be called manually – being included within ArcGlobe’s internal rendering process will assure that it is always called immediately after ArcGlobe’s internal drawing has completed (hence the event’s name, after-draw).

When to Create the OpenGL Portal

The `AfterDraw` event handler need only be created once, when the ArcGlobe globe itself is created. This will typically occur whenever the ArcGlobe-based application launches, so the developer needs to make sure that the `CreateOpenGLPortal()` method (or its equivalent) is called sometime during the application startup process. For embedded applications using the ArcGlobe Globe control, such a call can be included in the main application form’s constructor or its `Load` event handler.

However, there is one special case which necessitates that the `AfterDraw` event handler be recreated. This special case occurs whenever the internal globe is replaced (when a new Globe document (*.3dd) is loaded. When this occurs, the `AfterDraw` event handler must be recreated. This can be achieved by adding an event handler for the Globe control’s `OnGlobeReplaced` event, and then in the event handler body calling the `CreateOpenGLPortal()` method.

Finalization

One last point that needs be brought up with regard to integrating custom OpenGL rendering into ArcGlobe, corresponds to scene finalization. As mentioned during the

discussion of the SMURF method in the previous section, the scene must be finalized once rendering is completed for each frame. It was mentioned that when OpenGL is used in double-buffer configuration (the standard for 3-D), scene finalization is accomplished via a call to the `glSwapBuffer()` function.

However, as was the case with initialization, ArcGlobe takes care of calling this function internally as part of its internal rendering process – but only after any calls to `AfterDraw` event handlers have returned (*after the after-draws*). Thus, the developer should not call this function manually.

Practical Applications

Having now reviewed all the essential components and concepts necessary to build custom OpenGL rendering into ArcGlobe, it is now time to see the various pieces assembled in the context of practical applications. The explanation of such sample applications is the purpose of this section.

Sample Application Functional Details

Features

The sample application developed in support of this paper and referenced throughout is a simple satellite viewing application that shows the path, motion, and sensor coverage of a satellite orbiting the earth. The orbit path, direction, speed, and sensor coverage range of the satellite are fully configurable by the user. The three dimensional earth is provided via the Globe control, and all of the aforementioned satellite features are rendered using OpenGL.

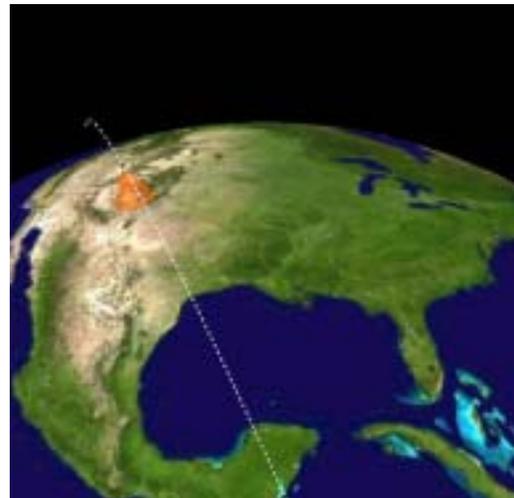


Figure 3.
ArcGlobe/OpenGL Sample Application

Platform

The sample application was developed for the Windows 2000/XP platform using Visual Studio .NET 2003. The C# programming language was used for all source code, and the version of ArcGIS Engine used was the 9.0 pre-release (January 2004) included in the 1.0 C/JMTK. To perform the OpenGL rendering in C#, CsGL - an open source wrapper around the native C/C++, OpenGL libraries are used. For more information on CsGL, consult csgl.sourceforge.net.

Sample Application Technical Details

As previously discussed, the sample application renders three animated OpenGL-based entities in ArcGlobe—a satellite, its sensor coverage, and its orbital path around the earth. Each of these entities is represented in the sample application source code by a corresponding C# class. The application's main form (which also holds the ArcGlobe Globe control) creates one instance object of each of these classes, then renders them in its `OpenGLRender(...)` method:

```
protected void OpenGLRender(ISceneViewer viewer)
{
    ...

    orbit.Render(...);

    ...

    sensor.Render(...);

    ...

    satellite.Render(...);
}
```

As can be seen from this excerpt, getting OpenGL-based code to work in an ArcGlobe context is really quite simple. To see how the SMURF method for rendering with OpenGL works in a practical context, the next subsections will discuss the rendering source code for the satellite itself and the orbital path. The satellite's sensor coverage, which is implemented in a way very similar to the satellite itself, will not be covered.

Rendering the Orbital Path

The following is the source code used to render the satellite's orbital path (line numbers added for convenience):

```
1     public void Render(double radius, double xTilt, double yTilt)
2     {
3         // push the current color attribute & set the color
4         GL.glPushAttrib(GL.GL_CURRENT_BIT);
5         GL.glColor3d(1, 1, 1);
6
7         // push the line attribute & set line stipple, style, and thickness
8         GL.glPushAttrib(GL.GL_LINE_BIT);
9         GL.glEnable(GL.GL_LINE_SMOOTH);
10        GL.glEnable(GL.GL_LINE_STIPPLE);
11        GL.glLineStipple(1, (ushort)0x0F0F);
12        GL.glLineWidth(m_lineThick);
13
14        // push the current modelview matrix & tilt the axis
15        GL.glPushMatrix();
16        GL.glRotated(xTilt, 1, 0, 0);
17        GL.glRotated(yTilt, 0, 1, 0);
18
19        // begin rendering a line
```

```

20     GL.glBegin(GL.GL_LINE_STRIP);
21
22     double x, y, radians;
23     for(double i = 0; i < 361; i++)
24     {
25         // convert from degrees to radians
26         radians = i / 180 * Math.PI;
27
28         // calculate the next vertex
29         x = radius * Math.Cos(radians);
30         y = radius * Math.Sin(radians);
31
32         // render the vertex
33         GL.glVertex3d(x, y, 0);
34     }
35
36     // end line rendering
37     GL.glEnd();
38
39     // pop the current modelview matrix
40     GL.glPopMatrix();
41
42     // pop the line attribute
43     GL.glPopAttrib();
44
45     // pop the current color attribute
46     GL.glPopAttrib();
47 }

```

The following list breaks down the line numbers by their corresponding step in the SMURF method:

- **S**ave the current state of any settings to be modified.
 - Lines 4, 8, and 15.
- **M**odify the state of appropriate settings.
 - Lines 5, 9, 10, 11, 12, 16, and 17.
- **U**se rendering primitives to build the scene.
 - Lines 20, 33, and 37.
- **R**estore the prior state of any modified settings.
 - Lines 40, 43, and 46.
- **F**inalize the scene by performing a buffer swap.
 - N/A—handled internally by the Globe control.

It should be noted how each line number under step ‘S’ of the SMURF method has a corresponding step under the ‘R’ step. Also note that it is not necessary for every ‘S’ line to be executed before any ‘M’ lines can occur. That is, it is not necessary for *every* setting whatsoever that will be modified to be saved before any modifications can be done—notice how the ‘M’ lines are interspersed with the ‘S’ lines. What is necessary, however, is that before any setting is modified, the corresponding ‘bit’ for that group of settings is saved (see lines 8 through 12, where the single `GL_LINE_BIT` actually stores four separate settings).

As can be seen, using an organized method (such as SMURF) to structure OpenGL rendering code can greatly simplify usage of an otherwise somewhat complex API.

Rendering the Satellite

The following is the source code used to render the satellite sphere (again, line numbers added for convenience):

```
1     public void Render(double degrees, double satRadius,
2         double xTilt, double yTilt)
3     {
4         // convert degrees to radians
5         double radians = degrees / 180 * Math.PI;
6
7         // calculate the orbital position of the satellite
8         double x = satRadius * Math.Cos(radians);
9         double y = satRadius * Math.Sin(radians);
10        double z = 0;
11
12        // push the current color attribute & set the color
13        GL.glPushAttrib(GL.GL_CURRENT_BIT);
14        GL.glColor3d(0, 0.4, 0);
15
16        // push the current modelview matrix, tilt & translate
17        GL.glPushMatrix();
18        GL.glRotated(xTilt, 1, 0, 0);
19        GL.glRotated(yTilt, 0, 1, 0);
20        GL.glTranslated(x, y, z);
21
22        // push the polygon bit, then set the polygon mode
23        GL.glPushAttrib(GL.GL_POLYGON_BIT);
24        GL.glPolygonMode(GL.GL_FRONT_AND_BACK, GL.GL_FILL);
25
26        // render the sphere using a quadric
27        GLU.gluQuadricDrawStyle(quadric, GL.GLU_FILL);
28        GLU.gluSphere(quadric, size, 32, 16);
29
30        // pop the polygon bit
31        GL.glPopAttrib();
32
33        // pop the current modelview matrix
34        GL.glPopMatrix();
35
36        // pop the current color attribute
37        GL.glPopAttrib();
38    }
```

The following list breaks down the line numbers by their corresponding step in the SMURF method:

- **Save** the current state of any settings to be modified.
 - Lines 12, 16, and 22.
- **Modify** the state of appropriate settings.
 - Lines 13, 17, 18, 19, 23, and 26.

- **U**se rendering primitives to build the scene.
 - Line 27.
- **R**estore the prior state of any modified settings.
 - Lines 30, 33, and 36.
- **F**inalize the scene by performing a buffer swap.
 - N/A—handled internally by the Globe control.

It should be noted that as mentioned in an earlier section, using quadric GLU objects to render spheres in OpenGL greatly simplifies the rendering process. From the listing, it can be seen that, using a quadric, the actual rendering of the sphere itself is confined to a single line (line 27) of code.

The only additional item that a developer should keep in mind when using quadrics is that quadrics, like any object, have to be created. To create a quadric, the `gluNewQuadric()` function is provided as part of GLU:

```
protected CsGL.OpenGL.GLUquadric quadric = GLU.gluNewQuadric();
```

Real-World Application—TOC 3-D

As shown in the last section, OpenGL can be readily integrated within the ArcGlobe framework to produce high performance and highly customized visualization for practical applications. The sample application developed was meant only as an example of what can be done in a relatively short amount of time leveraging only the bare basics of OpenGL functionality.

However, when used in tandem, ArcGlobe and OpenGL are capable of much more interesting and more powerful features than those covered in this paper. In it's work on the TOC 3-D initiative (introduced in an earlier section), *CTC* has developed a very high speed rendering library for the display of Military (MIL-STD-2525B and NTDS) symbology within ArcGlobe. Figure 4 demonstrates some of the capabilities of this rendering library.

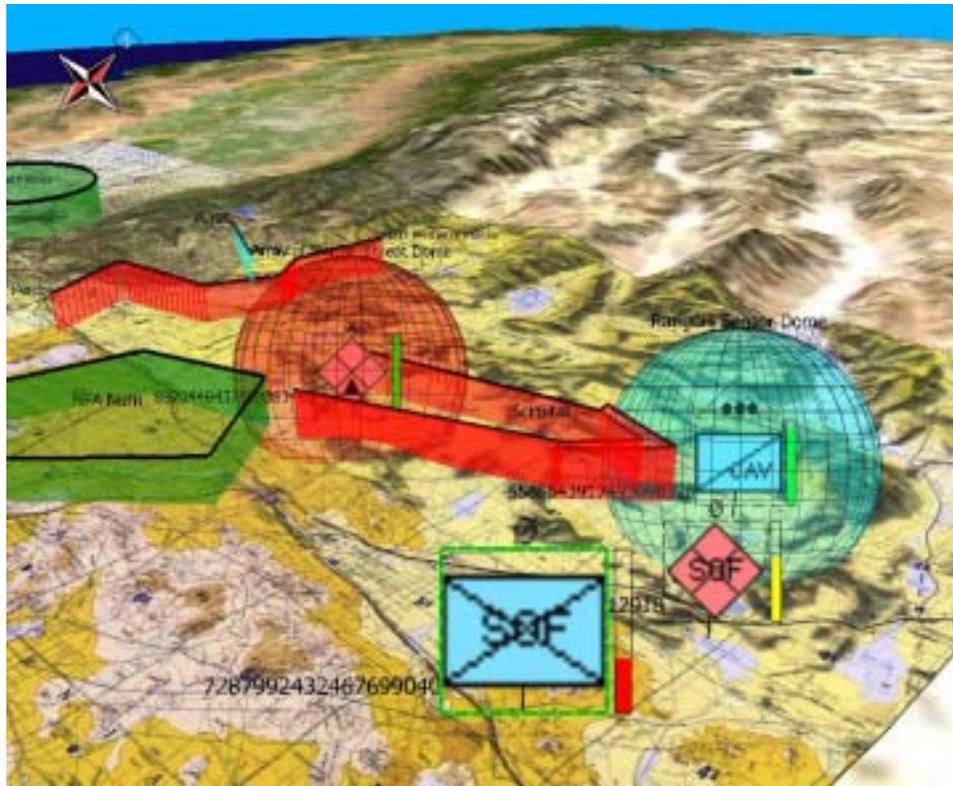


Figure 4. TOC 3-D Custom OpenGL Rendering in ArcGlobe

The TOC 3-D Rendering and Visualization Engine for OpenGL (RavenGL) takes full advantage of the possibilities that exist when ArcGlobe and OpenGL are used together. These two technologies have been combined to provide many advanced rendering features including:

- **Billboarding.**
 - Tactical symbols always rotate to face the camera (the user), even when the global scene is being rotated, zoomed, or otherwise animated.
- **Graphic Extrusion.**
 - Tactical graphics are drawn above the surface of the earth, and then the intervening area is filled semi-transparently with the graphic's affiliation color.
- **Lollipopping.**
 - Tactical symbols can be raised to a user-determined "eye level" on posts, while the bottom of the posts are still anchored to the symbols' true locations.
- **Sensor/Threat Domes.**
 - The effective sensor and weapons ranges of battlespace entities can be displayed as semi-transparent spheres.
- **Strength Thermometers.**
 - The relative combat power of each entity can be displayed next to the corresponding billboarded symbol.

The advanced features incorporated into the RavenGL stand as a strong testimonial to the flexibility and collaborative power of ArcGlobe and OpenGL.

Conclusion

ArcGIS provides numerous methods for extending, enhancing, and integrating functionality. Along with the standard mechanisms provided for extending ArcGIS, 3-D Analyst provides the additional capability to perform highly customized 3-D rendering using OpenGL.

This paper demonstrated how to integrate OpenGL-based custom rendering solutions into 3-D Analyst. It explained OpenGL development and then showed how to integrate OpenGL-based software components into 3-D Analyst products such as ArcGlobe. Finally, the concepts presented were shown working together using several sample and real world applications.

Acknowledgments

The TOC 3-D Program Team would like to acknowledge the support of PM GCC2 and specifically the members of the Technical Management Division (TMD). Additionally, several individuals have been instrumental in obtaining support and feedback of the TOC 3-D C/JMTK integration task. These individuals include:

Brett Cameron, Northrop Grumman Information Technology (NGIT) – TASC

John Day, ESRI

James Eakin, National Geospatial-Intelligence Agency (NGA)

Brady Hoak, ESRI

Susan Marchant, NGA

Gary Scoffield, ESRI

Clark Swinehart, ESRI

Mike Werling, NGIT - TASC

Appendices

Appendix A - Additional Resources

Developers who are interested in learning more about the material covered herein should consult the following resources:

- “OpenGL Game Programming.”
 - Prima Publishing, 2001.
 - ISBN #0761533303.
- “Beginning OpenGL Game Programming.”
 - Premier Press, 2004.
 - ISBN #1592003699.
- CsGL Documentation.
 - <http://csgl.sourceforge.net>.
- ESRI ArcEngine Documentation.
 - <http://arcgisdeveloperonline.esri.com>.

For those with further ArcGIS and OpenGL-related technical needs, *CTC* also offers the following:

- Custom ArcGIS, C/JMTK, and OpenGL Development/Integration.
- Training Courses.
 - ArcObjects for the C/JMTK.
 - Programming with OpenGL.

Appendix B – Obtaining, Building, and Running the Sample Application

Obtaining the Sample Application

The source code to the sample application may be obtained by emailing the authors. For author contact information, one should consult the section [Author Contact Information](#). The compressed file containing the software source and support information is approximately one megabyte.

Building the Sample Application

In order to build the sample application described in this paper, the following development and runtime environments are required:

- Windows 2000/XP platform.
- Visual Studio .NET 2003.
- ArcGIS Desktop or Engine 9.0.
- 3-D Analyst 9.0.

A README file with detailed build instructions is provided with the sample application. In general, only the following steps are needed:

- Create or obtain a valid ArcGlobe document (*.3dd).
- Open the “GlobeSat.sln” file in Visual Studio .NET 2003.
- Open the “GlobeSatForm.cs” file in Design view.
- Open the properties for the Globe control and set the “Globe Document” property to a valid ArcGlobe document (*.3dd).
- Once this step has been completed, the solution should build and execute without any further modification.

Author Contact Information

Jeremiah Montgomery, Software Engineer, *CTC*, montgomj@ctc.com, 814-269-6484

Christopher Moore, Software Engineer, *CTC*, moorec@ctc.com, 814-269-6232