THE THREE-DIMENSIONAL EVENT-DRIVEN GRAPHICS ENVIRONMENT (3D-EDGE)

By Jeffrey Freedman, Roger Hahn and David M. Schwartz

Stanford Telecom
7501 Forbes Blvd
Seabrook, MD 20706
(301) 464-8900

ABSTRACT — Stanford Telecom developed the Three-Dimensional Event-Driven Graphics Environment (3D-EDGE) for NASA Goddard Space Flight Center's (GSFC) Communications Link Analysis and Simulation System (CLASS). 3D-EDGE consists of a library of object-oriented subroutines which allows engineers with little or no computer graphics experience to programmatically manipulate, render, animate, and access complex three-dimensional objects.

I. INTRODUCTION

3D-EDGE was developed to allow programmers with little or no computer graphics experience to incorporate three dimensional solid objects into their programs. Other programmatic graphic interfaces [1,2] such as PHIGS require the user to have an in-depth knowledge of the type of object being modeled and how it is manipulated. This limits programmatic access of three-dimensional objects to people who have three dimensional computer graphics training. 3D-EDGE, on the other hand, uses very simple commands to manipulate, access and render the solid model. The user only needs to learn a few 3D-EDGE commands to use any three dimensional object because 3D-EDGE has the same generic interface for every object. This allows the user to access objects without knowledge of their internal data structure.

II. OVERVIEW

3D-EDGE can incorporate a wide variety of solid model representations, keeping their internal structures invisible to the user. Thus, once a user is familiar with 3D-EDGE, he/she can manipulate, render, animate and access any object regardless of its internal configuration.

Figure 1 shows the 3D-EDGE data hierarchy. The figure shows the 3D-EDGE data dependencies, and the supporting subroutines for each data type. The remainder of this section describes Figure 1 in detail.

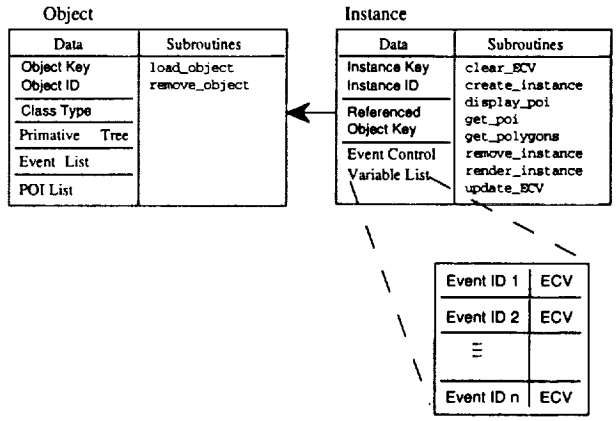

Fig. 1. 3D-EDGE Data Hierarchy Diagram

The database contains a description of a three dimensional object, an associated list of *events,* and a set of Points of Interest, (*PoI*'s, these are discussed below). *Events* control location, orientation, and any other configurable features of the model. For example, CLASS's model of the Space Shuttle contains *events* which control its location, orientation, the percentage Shuttle doors are opened, and the gimbal angles of the antennas. The Shuttle doors are opened by changing an appropriate *event's* value (e.g. "DOORS OPEN" *event*).

An *instance* contains a list of *Event Control Variables* (*ECV*'s) that specify values for every *event.* Since the user never modifies the object itself, only the *instance*, multiple *instances* of the same object can be controlled simultaneously

This work was supported by NASA Goddard Space Flight Center under constract NAS 5-31500.

while maintaining data integrity and minimizing memory usage. The user updates the *ECV*'s through a set of 3D-EDGE subroutines. The configuration of the instance is calculated only when the user either renders (graphically displays the *instance* of an object) or accesses information from the *instance*. Dynamic motion and animation can be effected by interactively changing *ECV*'s and re-rendering.

The user accesses information about an *instance* of an object by calling an appropriate query routine. The query returns to the user information which may include, but is not limited to, a polygonal description of each surface on the object, a surface color, a surface dielectric constant, and a *PoI*. A *PoI* is a location and direction on an object which moves with the object. For example, a *PoI* can be the location and direction of an antenna boresight, which automatically moves with the antenna.

Figure 2 shows an overall flow diagram of a 3D-EDGE program. The user first loads an *object* from the database. Then one or more instances of the *object* are created. The instance is modified by using the update_ECV subroutine, which changes a particular ECV in an *instance*. For example, an update_ECV can be used to open Shuttle doors or to gimbal antennas. Next, either information about the *instance* (e.g. polygonal locations or dielectric constants) may be accessed or the *instance* rendered. This process can then be repeated or the *object* is disposed of if it is no longer required.

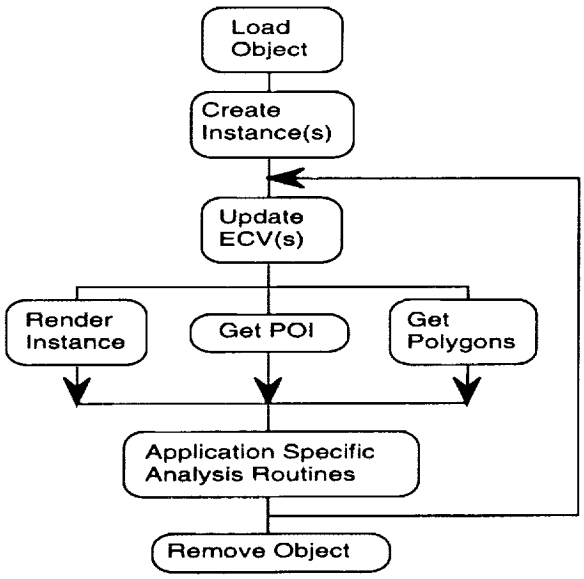

Fig. 2. 3D-EDGE Flow Diagram

III. OBJECTS

An *object* refers to a three dimensional model. 3D-EDGE is designed to allow easy programmatic access to these *objects*. Each *object* belongs to a particular *class*. The *class* refers to the underlying solid model description of the *object*. For example, the *polygon class* (currently the only *class* supported) refers to objects made from polygonal surfaces. Additional proposed classes of objects include descriptions based on extrusion, B-spline and fractal solid model representations.

A. Hierarchy

Objects are comprised of a set of hierarchically related components known as *primitives*. *Primitives* are related to one another through transformation matrices. An example of this hierarchical structure is a simplified Space Shuttle *object* which is comprised of 8 *primitives*: a fuselage, a nose, a tail, two cargo bay doors, and a robotic arm comprised of three parts: a turret, forearm and clamp, as shown in Figure 3. The fuselage is the *root primitive* which has five children: the nose, tail, doors and turret. The shoulder in turn has one child, the forearm, and the forearm one child, the clamp.

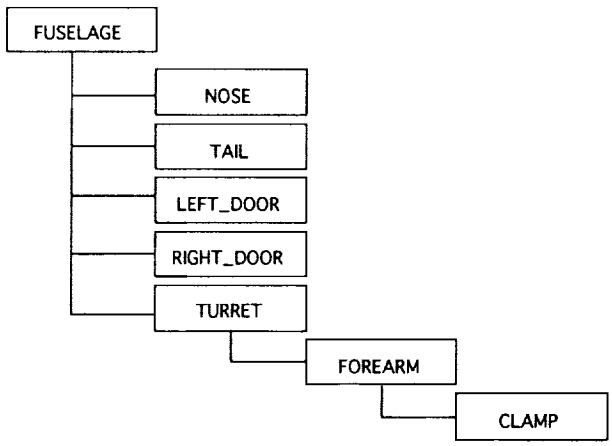

Fig. 3. Simple Shuttle *Primitive* Hierachy

The use of a hierarchical structure simplifies the model definition and provides a natural mechanism for specifying the manipulation of the model.

If an *event* is invoked to rotate the turret of the robotic arm, the rest of the arm moves as well. This is true for all parent-child pairs. As the parent moves, so do the children and the children's children, on down the line. Thus, by rotating an *object's root primitive* the entire *object* is effectively rotated in space.

B. Attributes

Attributes are used to define an *object's* physical characteristics. Examples of attributes include color, reflective coefficient, and dielectric constant. An *object* inherits a set of attributes from 3D-EDGE based upon the *object's class*. An *object's attributes* are defined within the *object's database*.

C. Inheritance

Inheritance, an important feature in 3D-EDGE, takes two forms: *events* and *attributes*. Both are inherited from more general levels to more specific levels. For example, if an *object's root primitive* is defined to be "white" in the *database*, then the rest of the *object* is considered to be "white." However, if a particular *primitive* has "blue" as the value of its *color attribute*, that *primitive* and any of its descendants will also be "blue" unless they too re-specify the *color attribute*.

D. Points of Interest

A *Point of Interest* ("*PoI* ") is a location and direction on a *primitive* which moves with the *primitive*. By using 3D-EDGE's query language, the location and direction of the *PoI* may be retrieved at run-time. This allows programmatic use of the information. A program developed for CLASS uses a *PoI* to represent the antenna beam (boresight) of a satellite's gimbal antenna. As the antenna moves, so does its boresight. The *PoI* information is then used to control the camera (the user's view) in a real-time graphics package. The scene is then viewed from the antenna along its boresight.

IV. EVENTS

Each object has an associated list of *events*. *Events* are used to control and manipulate the *object*. They control location, orientation and any other variable attributes of the model. *Events* are characterized by the following parameters: Type, Level, and Order.

This concept of *events* lies at the heart of 3D-EDGE because it means that the user needs to know nothing about how the *object* is defined in order to manipulate it. From the user's perspective, the type or level of the *event* is irrelevant. The *event* need only be defined for the *object* and invoked by the user; 3D-EDGE takes care of the rest.

A. Event Types

Events may be either simple or complex. For example, CLASS's model of the Space Shuttle contains *events* which control its attitude (orientation in space), the percentage the Shuttle doors are opened, and the gimbal (rotation) angles of the antennas. A simple event on the Shuttle would be a "ROLL". In real terms this means a rotation of the Shuttle about its positive x-axis. A complex *event* would be "OPEN_DOORS" which it entails rotating two different *object* components about their respective rotational axes.

B. Event Levels

Events are defined at three different levels: the *object* level, the *class* level and the *system* level. *Object* level events are *object* -specific and as such are defined in the *object's* database. *Class* level events are those that are defined within 3D-EDGE code but only for a specific *class* of objects. *System* level events are events which are defined in 3D-EDGE system code for all objects. The "ROLL" *event* noted above is a *system* level event as it is defined for all objects. However, the "OPEN_DOORS" event is an *object* level event defined only for the Shuttle.

C. Event Order

Computer graphics algorithms typically use translation and rotation matrices to manipulate objects. These matrices must be applied in a set order to achieve the desired effect. For example, a translation issued before a rotation will place an object at a different location than the same rotation issued before the translation.

In 3D-EDGE, the user can specify events in any order. However, the order in which the

events are actually invoked is defined by the 3D-EDGE system. An example of this is the way Shuttle antenna gimbal angles are specified. Two separate angles are necessary to define the position of a gimballed antenna: azimuth and elevation. Mathematically, the order in which the transformations are performed on the *object's* data points is relevant. Therefore, within the database for the Shuttle the *events* "SET_AZIMUTH" and "SET_ELEVATION" are defined such that they will be performed in the appropriate order at run-time. However, the user can update the ECV's in any order and achieve the same results.

D. Event Control Variables (*ECV*)

ECV's are used to control events. The *ECV's* are specified as either an explicit value or as a percentage of the event's range. How the *ECV* is to be specified is *event*-dependent. An example of an *event* requiring an explicit value is the "SET_AZIMUTH" *event* defined above. When invoking the *event*, the user could specify an ECV of 240, which would mean "rotate the antenna about the appropriate axis two-hundred forty degrees." The "OPEN_DOORS" *event* for the Shuttle is an example of an *event* whose *ECV* is to be specified as a percentage. When invoking the *event,* the user could specify an *ECV* of 50, which would mean "open the Shuttle bay doors halfway." The full range of motion of the doors is defined in the *object's* database.

V. INSTANCES

The user creates an *instance* of an *object* in order to control, access and render the *object* at run-time. The *instance* and its associated subroutines are illustrated in Figure 1. An *instance* of an *object* consists of a pointer to the instance (called an *instance key*), an *instance* identifier, a pointer back to the *object*, (called an *object key*), and a set of *ECV*'s for all *events* defined for the *object*. It is the *instance* that is used to either render an object or access information about the object.

Since the user never modifies the *object* itself, multiple *instances* of the same *object* can be controlled simultaneously while maintaining data integrity and minimizing memory usage.

VI. SUBROUTINES

Subroutine calls are the vehicle through which the 3D-EDGE system routines are accessed. There are subroutines for loading *objects*, invoking *events*, and querying for information about *objects*. Interfaces to the subroutines are available for both C and FORTRAN. One of the subroutines that allows the user to get information about the object is "get_polygons". The "get_polygons" subroutine returns to the program the transformed data points describing the current configuration of an *instance* of an object. By passing a mask which describes the information to be extracted, "get_polygons" can be used to access other information about the object like *dielectric constants* and *color*. Although these attributes usually remain constant for the *object* at any configuration, it is often useful to access this type of information.

VII. EXAMPLE PROGRAM

Figure 4 contains a sample program. The program first loads the Space Shuttle object by passing the name of the database containing a model of the Space Shuttle, "shuttle_file," to the "load_object" subroutine which then returns an integer *object key*, "object_key." "object_key" is then used by the "create_instance" subroutine to create two different *instances* of the Shuttle. "create_instance" is invoked by passing to it the "object_key," specifying an "instance_id" ("DISCOVERY" or "COLUMBIA" in this case), and specifying a load preference. An integer *instance key*, used to reference the instance in the remainder of the program, is then passed back.

Once the two *objects* are *instantiated* the program loops 100 times changing the variable i from 0 to 99. Within the loop, "DISCOVERY", identified by its *instance key*, "Discovery_key," and "COLUMBIA," identified by its *instance key*, "Columbia_key," have their configurations altered by invoking specific *events*. Specifically, "DISCOVERY" has its doors opened, is pitched, and is yawed by i, i/5, and i respectively. "COLUMBIA" only has its doors opened i*2 percent of their range of motion. (Note: Since the "OPEN_DOORS" event is defined with "HARD_LIMITS" of $0 < x < 100$, for any value

```
/* LOAD OBJECT */
object_key = load_object(shuttle_file);
/* INSTANTIATE SHUTTLE */
Discovery_key = create_instance(object_key,
    "DISCOVERY",LOAD_ABSOLUTE);
Columbia_key = create_instance(object_key,
    "COLUMBIA",LOAD_ABSOLUTE);
/* LOOP */
for (i=0;i<100;i++){
    update_ECV(Discovery_key,"OPEN_DOORS",i);
    update_ECV(Discovery_key,"PITCH",i/5);
    update_ECV(Discovery_key,"YAW",i);
    update_ECV(Columbia_key,"OPEN_DOORS",i*2);
    get_polygons(Discovery_key,VERTEX_NORMAL |
          DIELECT, polygon_points_array1,
          num_points1);
    get_polygons(Columbia_key,VERTEX_NORMAL |
          DIELECT, polygon_points_array2,
          num_points2);
    process_polygons(polygon_points_array1,
          num_points1);
    process_polygons(polygon_points_array2,
          num_points2);
}
```

Fig.4. Sample Program

of i or i*2 > 100, the doors will only be opened the maximum of 100 percent.) Once all of the *ECV*'s have been changed, subroutines are called which will cause the *instances'* configurations to be calculated. In this case, the subroutine is "get_polygons." This subroutine causes the polygons, their vertex-normals, and their dielectric coefficients to be passed back to the program. The first call calculates the points specifying the polygons for "DISCOVERY" and passes back "num_points1" points in the "polygon_points_array1" array. Similarly, the next call to "get_polygons" returns the "num_points2" points defining "COLUMBIA" in the array "polygon_points_array2." The program then calls its own routine "process_polygons" to do the actual analysis desired.

VIII. RESULTS

Figure 5a shows a model of the Compton Gamma Ray Observatory (GRO) satellite which was incorporated into 3D-EDGE. 3D-EDGE can render GRO, move it, change its orientation, gimbal the antennas, and rotate the solar panels. The solar panels are rotated by first creating an *instance* of GRO, and then updating the "ROTATE_PANELS" *EVC*. Figure 5b is the GRO satellite after the solar panels have been rotated.

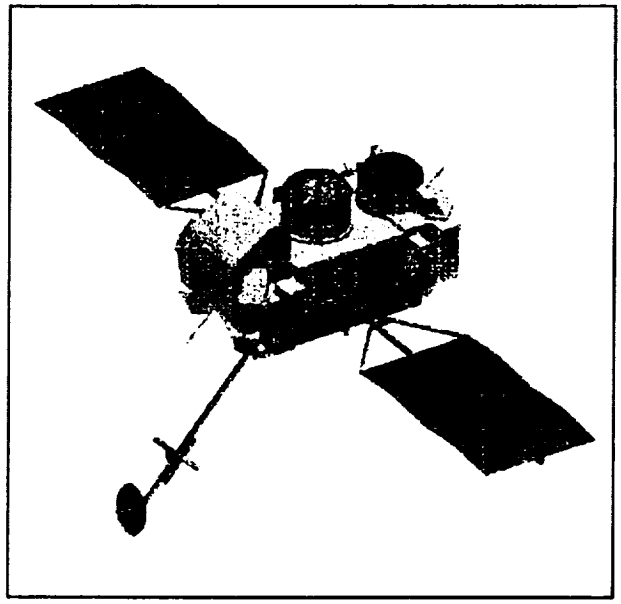

Fig. 5a. GRO before "ROTATE_PANELS"

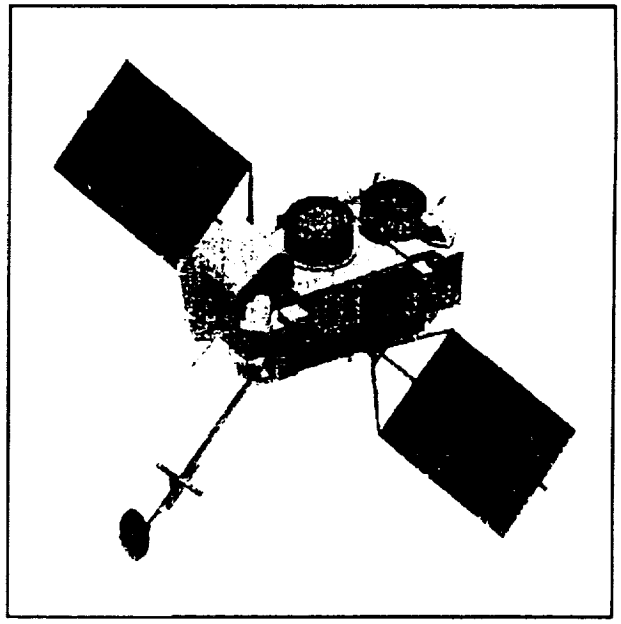

Fig.5b. GRO after "ROTATE_PANELS" by +50 Degrees

CLASS uses 3D-EDGE in several of its programs to manipulate and render solid models. The CLASS Multi-Path Program (MPP) currently uses 3D-EDGE to control a three dimensional description of the spacecraft it is analyzing. The MPP has a minimal software interface

to 3D-EDGE because a single query of the model returns the necessary information (the dielectric constants and locations for each surface). The 3D-EDGE generic interface also allows the MPP to interchangeably use any spacecraft. Additionally, the CLASS Flight Performance System (FPS) uses 3D-EDGE for a graphical display of multiple models during a simulated Shuttle mission.

Before the development of 3D-EDGE, CLASS analysis programs that used solid models had severe limitations. These limitations included long development times, non-portable applications, lack of solid model data integrity between programs, and programs that could not interchangeably use different objects. 3D-EDGE solves these problems with an easy-to-use standardized graphics environment.

IX. CONCLUSION

3D-EDGE was designed on the principle that it is more important and efficient to spend time in the definition of a three-dimensional model than in the incorporation of that model into software. 3D-EDGE requires that when an object's database is being developed the events be defined along with the solid model description of the object. However, once defined, an object can be used by anyone with a knowledge of 3D-EDGE, even if they have minimal knowledge of 3D graphics. Further, by using abstract events and classes of data, virtually anything can be modeled and manipulated using only a small set of subroutines.

REFERENCES

[1] T.L.J. Howard, W.T. Hewitt, R.J. Hubbold, K.M. Wyrwas, *A Practical Introduction to PHIGS and PHIGS PLUS*, Addison-Wesley Publishing Company, 1991.

[2] *Starbase Graphics Techniques*, Hewlett Packard Company, 1991

[3] N. Magnenat-Thalmann , and D. Thalmann, *New Trends in Animation and Visualization*, John Wiley and Sons, New York, 1991

[4] J. Freedman, *Video Compression*, Ph.D. Dissertation, University of Maryland, May, 1991

[5] J. Freedman, T. Kaplan, "Layered Command Driven Animation," Proceedings of SIGGRAPH 1992, July 1992 (In Review)

[6] T. Kim, "Technical Reference for C.L.A.S.S. Vehicle Multipath Modeling Program," Stanford Telecommunications, Seabrook MD, May, 1991

[7] C. S. Park, *Interactive Microcomputer Graphics*, Addison-Wesley Publishing Company, 1985

[8] T. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley Publishing Company, 1991

[9] J. Freedman, R. Hahn, D. Schwartz, "The Three Dimensional Event Driven Graphics Environment (3D-EDGE)," IEEE National Telesystems Conference, May 1992.