EXPLOITING CLASSES IN MODELING AND DISPLAY SOFTWARE

Turner Whitted and Eric Grant

Department of Computer Science
The University of North Carolina
Chapel Hill, North Carolina

Abstract

The class concept is one component of object-oriented programming systems which has proven useful in organizing complex software. In experimenting with the use of classes for geometric modeling applications, we have devised a class hierarchy that yields some conceptual order in the midst of diverse representations of shapes. Rather than searching for a uniform primitive representation, we accept the diversity and build a framework in which dissimilar models are combined in an orderly manner.

KEYWORDS: geometric modeling, procedure models, object-oriented programming

Introduction

Geometric modeling systems can become extremely complex when design applications demand flexibility in the representation of shapes. A major challenge for programmers who create such systems is to preserve order in spite of this complexity. Trends in this direction include moves toward uniform data representation and very general mathematical representations of shapes. However, the advent of specialized procedural modeling techniques is a step away from uniformity which strains programmers' abilities to cope with the diversity that it presents. We feel that using an object-oriented programming methodology helps to solve this problem.

In spite of the recent interest in object-oriented programming, we have seen only a few published examples of 3-D graphics systems built in an object-oriented environment [Hedelman, Lorenson]. For the most part, the examples that we have seen emphasize the message passing aspects of Smalltalk-like languages and do not pay much attention to the effective specification of classes.

In this paper we describe the class hierarchy of an experimental modeling and display system which we have assembled in order to study the problems of constructing extremely complex geometric objects. We emphasize identifying common elements of geometric procedures which can be shared among representations. Our guiding principle is that methods should be shared by as many classes as possible and that they should belong to classes as high up in the class hierarchy as possible. In a following section we describe our attempt to define a class hierarchy that meets this criterion.

Diversity of Geometric Representations

Geometric models used in computer graphics have become so diverse that they don't seem to fit any rational scheme of classification. There has been a substantial amount of development of modelers that manipulate polygonal meshes, models that produce parametric surfaces or algebraic surfaces, and unified modeling systems that handle all three representations. Some of these have gone so far as to devise a common representation to ease the difficulty of storing and manipulating the diverse representations.

The widespread use of procedure models [Newell] has complicated the issue even further since the procedures are often restricted to such narrow purposes as creating trees [Bloomenthal], terrain [Fournier], or grass [Reeves]. To be sure, there are general purpose procedures such as sweeping, and there are generalizations such as graftals [Smith] which provide a common framework for a variety of individual geometric procedures.

The common simplification of reducing all complex shapes to polygonal approximations is no longer feasible when the number of primitive elements exceeds a few tens of thousands. Modification of such complex collections by users is nearly impossible. Common operations such as interference checking and display are confronted with massive amounts of data in this case.

Classes

One of the more successful mechanisms for coping with the complexity of programming is the use of classes. Originally a feature of the Simula language, classes are a central feature of Smalltalk [Goldberg]. Their value is more apparent when one considers that mature languages such as Lisp [Cannon] and C [Stroustrup] have been extended to include classes.

Classes are defined by a set of instance variable declarations and a collection of methods [Robson]. Objects are instances of a class. In a procedural geometric model, the model itself is an object.

The object-oriented organization gives a programmer the benefits of encapsulation and inheritance. To us, the more important property is the inheritance of methods and instance variable declarations. This means that objects that differ only slightly can be cast as members of distinct subclasses of the same superclass. Shared methods and instance variable declarations belong to the superclass. This code sharing has the beneficial side effect of reducing the overall code size. More importantly, development effort is reduced since programmers don't spend time writing the same methods over and over.

Evolution of the Intelligent Modeler

The overall goal of our research is to create detailed geometric models from sparse non-geometric inputs. We share this goal with several similar projects [Feiner, Holynski, Friedell]. Our original intent was to produce a more or less conventional interactive modeling program with an geometric knowledge base as its central element. In operation, the modeler would interpret guidelines provided by the user to invoke rules in the knowledge base which would in turn generate geometric primitives.

This approach was abandoned when we recognized it as an immense, monolithic procedure model. We then settled on a divide and conquer approach to reduce individual components of the modeler to manageable proportions. This then raised the additional problem of coordinating the actions of a multitude of autonomous procedures. For recursive procedures, such as subdivision, which maintain a geometric hierarchy as a product of their operation, we include methods by which mutually constrained objects agree on how each affects the other [Amburn]. So far we have no methods to satisfy mutual constraints for non-recursive procedure models.

Reducing the size of individual intelligent procedure models does not diminish the overall complexity of the modeler. For this we need the class hierarchy described in the following section.

Classes for a Modeling and Display Package

We have built (and continue to expand) an experimental modeling and display package based largely on generic procedure models. A description of the display system is included in this discussion because modeling operations play such an important role in its operation. Figure 1 shows the class hierarchy of the modeling portion of the package. While it may seem logical to organize the classes based on similar properties (i.e. one superclass for curved surfaces, another for polyhedra, etc.), our organization is based on common methods.

Diverse geometric representations can lead to a broad, flat organization of classes. However, our subtree for pro-

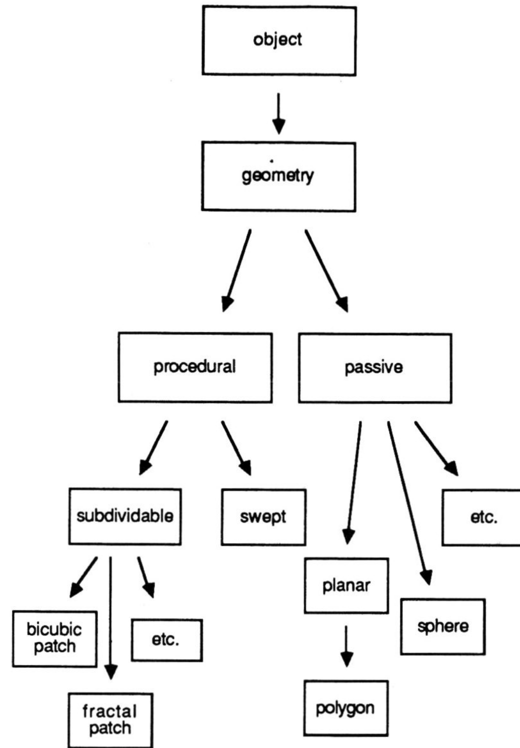

Figure 1. Geometry class hierarchy.

cedural geometry is five levels deep. This is achieved by factoring methods and placing the generic component as high in the tree as possible. Figure 2 is a condensed listing of four levels of one subtree (the *Object* class at the root of the entire class tree is omitted). At the lowest level is the *Bezier* subclass. Only methods specific to the *Bezier* subclass actually belong to it.

To illustrate how methods can effectively be shared, consider the class of subdividable surfaces (the lower three levels of figure 2). A typical display algorithm for this type of surface calls for recursive subdivision to a predetermined level of detail followed by the tiling of polygons formed from the mesh of points generated by the last subdivision stage (figure 3). The subdivision method belongs to the individual subclass. The tiling method, on the other hand, belongs to the class of subdividable surfaces instead of the individual subclasses. Likewise the test for level of detail belongs to the superclass rather than its descendants. This method inheritance is possible because the result of subdivision is a collection of vertices and neither the tiler nor the level of detail test care how the vertices were produced.

The display classes outlined in figure 4 are quite different from ones proposed early in the project. The initial hierarchy had fewer levels and far more classes than the final design. For example, we originally proposed a separate

```
// The class of all geometric elements
class Geometry
    {

    // GEOMETRIC INFORMATION

    int numVerts;
    vertex **v;
    BOOL bBoxSet;         // boolean for whether bBox is set
    boundingBox bBox;     // bounding box parameters

    matrix tform;         // transformation matrix

    // SURFACE PROPERTIES

    properties *color;    // surface color information

    // RENDERING INFORMATION

    rendParam rendInfo;   // rendering parameters (shading)
    float detail;         // level of detail

    // note: actual position information is defined by subclasses.
    // 'v' should be set up to point to position information.
    // This allows common methods to be implemented more easily.

public:

    // GEOMETRIC INFORMATION
    boundingBox getBBox();   // return bounding box info
    normal getNormal();      // return surface normal

    // TRANSFORM GEOMETRY
    void rotate(axis,float);
    void translate(float,float,float);
    void scale(float,float,float);
    void transform(matrix);    // transform according to supplied mat:


    void tile();               // generate rendering primitives

    void expand();             // expand geometry if possible
                               // for example: subdivide

    void collapse();           // reduce geometric complexity

    void textDumpGeometry();   // show geometric info in various forms
    void graphDumpGeometry();
    };


// The class of procedurally modeled geometry
class ProcedureGeometry: public Geometry
    {
    };


// The class of all subdividable procedure models
class SubGeometry: public ProcedureGeometry
    {
    int level;                      // number of times subdivided
public:
    void subdivide(float);
    BOOL terminationTest(float);    // level of detail test
    void changeBasis(int);
    void tile();
    };


// The class of subdividable bicubic Bezier patches
class Bezier: public SubGeometry
    {
    Bezier *parent;          // pointer to parent of this patch
    vertex cpoints[4][4];    // the control points
    float umin;              // u parameter range in original patch
    float umax;
    float vmin;              // v parameter range in original patch
    float vmax;
    Bezier *children[4];     // pointers to subpatches of this patch
public:
    void subdivide(float);   // redefine superclass methods
    };
```

Figure 2. Geometry class definitions in the C++ langauge with sample subclasses.

```
// expand: a generic subdivision method
// class:  SubGeometry

SubGeometry.expand()
{
        if (this->terminationTest(this->detail) == DONE)
                this->tile() ;
        else
                this->expandIt() ;
}


// expandIt: geometry specific subdivision
// class:     Bezier

Bezier.expandIt()
{
        // split current patch into four
        // subpatches and expand each
        // one in turn

        new patch1 ; patch1.f00(this) ;
        patch1.expand() ; delete patch1 ;

        new patch2 ; patch2.f01(this) ;
        patch2.expand() ; delete patch2 ;

        new patch3 ; patch3.f10(this) ;
        patch3.expand() ; delete patch3 ;

        new patch4 ; patch4.f11(this) ;
        patch4.expand() ; delete patch4 ;
```

Figure 3. A subdivision procedure divided into a generic method and a subclass specific method.

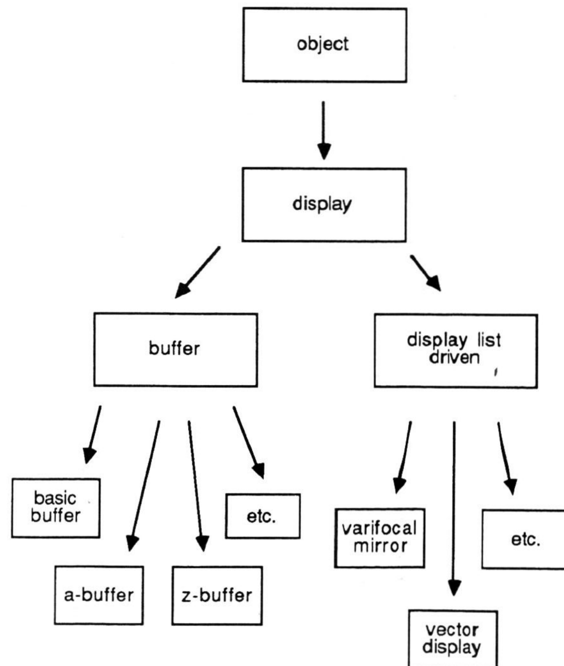

Figure 4. Display classes.

display class for BSP trees [Fuchs]. However, the BSP tree display algorithm is really a sorting method for geometric objects followed by a tiling method which deposits polygons in an image buffer. After factoring the algorithm into its constituent elements, and defining each element as a method of the most appropriate class, we can easily use the BSP tree sorting method in ray tracing to reduce the number of ray-surface intersection tests.

There are a number of commonly used algorithms, such as ray tracing, which are being ripped apart and reconstituted as methods belonging to various geometry and display classes. In general, as we recognize the common elements in different classes, the class tree becomes deeper and narrower.

Conclusion

This paper is not a sermon on the virtues of object-oriented programming. It is concerned instead with its application to modeling and display problems.

Organizing modeling representations into a class hierarchy has drastically altered the way we view geometric models. In particular, we have factored methods into generic and specific parts so that the generic parts can be shared. We have also found ourselves looking for ways to apply those methods that we have on hand to a broader range of geometric problems. We feel that our approach not only makes the programming of the modeling and display package more manageable, but it expands the range of features that the package can support.

Acknowledgement

Our colleague Phil Amburn is responsible for much of the work reported here. We thank Tim Rentsch for his review of the draft. This research is supported in part by Schlumberger-Doll Research.

References

[Amburn]
Amburn, Phil, Eric Grant, and Turner Whitted, "Managing Geometric Complexity with Enhanced Procedural Models," to appear in Proceeding of Siggraph '86.

[Bloomenthal]
Bloomenthal, Jules, "Modeling the Mighty Maple," *Computer Graphics,* vol. 19, No. 3 July 1985 pp. 305-311.

[Cannon]
Cannon, Howard, "Lisp with Flavors," Symbolics Inc., internal report.

[Carlson]
Carlson, Wayne, "An Advanced Data Generation System for Use in Complex Object Synthesis for Computer Display," Proceedings of Graphics Interface '82, May 1982, pp. 197-204.

[Feiner]
Feiner, Steven, "APEX: An Experiment in the Automated Creation of Pictorial Explanations," *IEEE Computer Graphics and Applications*, vol. 5, no. 11, November 1985, pp. 29-37.

[Fournier]
Fournier, Alain, Don Fussell, and Loren C. Carpenter, "Computer Rendering of Stochastic Models," *Communications of the ACM,* vol. 25, No. 6 June 1982 pp. 371-384.

[Friedell]
Friedell, Mark, "Automatic Synthesis of Graphical Object Descriptions," *Computer Graphics*, vol. 18, no. 3, July 1984, pp. 53-62.

[Fuchs]
Fuchs, Henry, Gregory D. Abram, and Eric D. Grant, "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics*, vol. 17, No. 3 July 1983 pp. 65-72.

[Goldberg]
Goldberg, Adele, and David Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, 1983.

[Hedelman]
Hedelman, Harold, "A Data Flow Approach to Procedural Modeling," *IEEE Computer Graphics and Applications*, vol. 3, no. 9, December 1983, pp. 18-25.

[Holynski]
Holynski, M., "Meaning Oriented Imaging for Graphics Interface," Proceeding of Graphics Interface '86.

[Lorenson]
Lorenson, William, "An Object Oriented Design of a Graphics Animation System," General Electric CR&D, Internal Report, 1984.

[Newell]
Newell, Martin, "The Utilization of Procedure Models in Digital Image Synthesis," PhD dissertation, Department of Computer Science, University of Utah, 1975.

[Reeves]
Reeves, William T., and Ricki Blau, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems," *Computer Graphics,* vol. 19, No. 3 July 1985 pp. 313-322.

[Robson]
Robson, David, "Object-Oriented Software Systems," *Byte*, vol. 6, no. 8, August 1981, pp. 74-86.

[Smith]
Smith, Alvy Ray, "Plants, Fractals, and Formal Languages," *Computer Graphics*, vol. 18, no. 3, July 1984, pp. 1-10.

[Stroustrup]
Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.