BABEL: A Generic Data structure for Geometric Modeling

Robert F. Tobler *Helwig Löffelmann* *Thomas Galla*
Werner Purgathofer

Institute of Computer Graphics
Technical University of Vienna

Abstract

We present a basic data structure for geometric data which can be adapted to represent common geometry representations like CSG, BSP, aso. The new data structure has been designed to be easy to use, and easy to extend. Due to the representation of geometric data using a directed acyclic graph, a number of the standard rendering algorithms can be used on the data structure in a very straightforward way. The new data structure has been implemented as a C++ library and can therefore serve as high-level tool for developing graphics applications, or as an extension for using C++ as a modeling language.

1 Introduction

There are a lot of common data structures in computer graphics which are based on a hierarchical representation of geometric data. When these data structures need to be implemented a lot of the code is similar, and can be shared, if the common features of these data structures were factored out. In this paper we present the structure of such an optimized representation of geometric data.

The main goals in developing this data structure were the following:

- provide all the basic mechanisms for implementing a geometric data structure
- provide an easy-to-use, high level programming interface for working with a geometric data structure
- provide an easy way to visualize the data structure using different rendering methods
- provide an object-oriented implementation of the data structure that can be easily extended

The new data structure has been implemented in C++ [Stro 91] and can be viewed as an extension to this language, since it provides high-level geometric data types for C++ (For this reason it was called BABEL: Building a Better Language).

2 The Structure of BABEL

2.1 The Geometric Primitives of BABEL

Since BABEL has to represent geometric data, we have to define the basic building blocks of our data structure. These are the geometric primitives we will use to build more complex geometric objects. These building blocks can be classified according to their dimension n:

- $n = 0$: vertex
- $n = 1$: line, general curve
- $n = 2$: plane, polygon, general surface
- $n = 3$: sphere, cube, cylinder, aso.

Obviously the set of objects that can be represented using a data structure depends on the geometric primitives that are available. It is therefore useful to foresee the possibility to extend the data structure by adding additional primitives. If an programmer needs to represent a lot of half-spheres in his application, it might be useful to add this class of objects as a new primitive to the geometric data structure. For this reason BABEL has been designed to be open for extensions by the user.

2.2 Operators to Build Complex Objects

In order to build more complex geometric models out of these primitives we need a way to combine them. We chose to use a directed acyclic graph (DAG) as a hierarchical representation of the geometric data. The leaf nodes of this graph are the geometric primitives explained earlier, and the intermediate nodes are operators, that combine, specialize, or even select these leaf nodes. In order to make it possible to represent standard data structures like CSG-trees and BSP-trees [Fuch 80] with BABEL, a number of operators had to be defined, e.g.:

- the CSG operators *and*, *or*, and *sub*, with two subnodes that represent the left and right operand
- the BSP separator with two subnodes that represent the positive and negative half-space, and a third subnode that represents the polygons contained in dividing plane
- a group operator to bundle primitives

Boundary representations are normally realized by storing a single instance for each vertex, and referencing these vertices by each polygon that shares these vertices. This can also be done in BABEL if we add grouping operators which define polygons. So in BABEL a simple polygon is a specialized group that contains its vertices as subnodes.

2.3 Attributes

A geometric data structure needs to hold more than just the geometric data: we want to represent things like color, material, and even names (for identification/selection issues) in our data structure. All these things can be thought of as attributes to the geometry in our data structure.

The standard way of storing these attributes, is as fields in the geometry representation. In order to make our data structure more flexible we chose a different way of representing attributes: each attribute is represented by a unary operator in the DAG. An attribute is valid for all of its subnodes. Within the DAG of subnodes an attribute of the same type can override the value of the currently valid attribute.

Some geometric properties may also be considered as attributes: take the normal to a surface as an example. If normals are added as attributes in the DAG, the data structure can support shading algorithms like phong shading [Buit 75].

2.4 Transformations

Although transformations have a profound effect on the geometry of an object they have also been implemented as attributes (unary operators). This makes it very easy for different objects to share some geometric information. Consider the example of multiple chairs of a single type in a room: there is only one representation of the geometry of a chair, but it is reused by multiple references using different transformations.

3 Manipulating BABEL Objects

The main purpose of a geometry data structure is the implementation of various geometric algorithms. In order to support a simple way of expressing these algorithms a number of manipulation methods have been implemented.

3.1 Graph Operations

A number of algorithms need to modify the DAG that represents geometric data. Due to the general nature of our data structure the common functionality of theses algorithms could be factored out and a number of tools could be provided, which reduce the aomount of code necessary for implementing these algorithms. The following tools for these operations have been implemented:

graph iterators: These iterators make it possible to iterate over a BABEL DAG in prefix, postfix, or infix order.

substitution methods: These methods allow the substitution of subtrees that satisfy given conditions.

tagging facility: This allows to allocate a tag bit for each node of the DAG, and thereby simplifies all algorithms that need to mark certain nodes. Some of the graph iterators depend on the functionality provided by this facility.

DAG optimizers: For various applications it is necessary to rearrange the BABEL DAG to achieve maximum preformance, e.g.: for raytracing the transformation nodes should be collected so that only the minimal number of necessary transformations remain in the DAG.

3.2 Storing and Retrieving

Of course a data structure is only useful, if it can be easily stored and retrieved. The design of BABEL includes a general scheme for storing and retrieving a DAG. Using a postorder walk of the DAG, each leaf node of the tree can be stored after all its subnodes have already be stored. This facilitates a very simple retrieval algorithm that can recreate all references to subnodes of a node in the tree, as soon as it has been retrieved. Since this algorithm is incorporated into the basic functionality of our data structure, implementing the storage and retrieval methods for single nodes is very simple, and consists mainly of storing and retrieving of all private data of the node.

In order to achieve maximal portability of the data, we decided to use a purely textual representation of the data. The syntax of this textual representation was chosen to be a very small subset of valid C++ (see the example below). This allows to include geometric data into the executable of an application, which may be useful to shield the application from manipulations.

Example of BABEL file format:

```
Solid getSolidTest()
{
  SolidArray s(11);
  int i = 0;

  s[0] = Solid::newVertex(Pnt3D(0,1,0));
  s[1] = Solid::newRGB(s[0],RGB(0,0,0));
  s[2] = Solid::newVertex(Pnt3D(0,0,2));
  s[3] = Solid::newRGB(s[2],RGB(1,1,1));
  s[4] = Solid::newVertex(Pnt3D(0,3,1));
  s[5] = Solid::newTriangle(s[1],s[3],s[4]);
  s[6] = Solid::newRGB(s[4],RGB(1,0,0));
  s[7] = Solid::newVertex(Pnt3D(0,2,3));
  s[8] = Solid::newRGB(s[7],RGB(0,1,0));
  s[9] = Solid::newTriangle(s[3],s[6],s[8]);
  s[10] = Solid::newGroup() << s[5] << s[9];

  return s[10];
}
```

3.3 Rendering

Without a decent way of rendering, a geometric data structure would be pretty useless. For this reason BABEL has been designed to easily support various different rendering algorithms. The hierarchical structure of the representation, is very natural for the following algorithms:

z-buffer [Catm 74]: Since all attributes are unary operators in the DAG, a depth first walk of the DAG in prefix order will produce all attributes and primitives in the correct sequence for rendering them into the z-buffer. Obviously this algorithm does not natively support CSG operators, therefore some CSG conversion algorithm must be applied to the DAG first.

raytracing [Appe 68]: A CSG raytracer [Roth 82] is already based on a hierarchical data structure and only needs to handle polygonal data as well in order to cope with the BABEL DAG

BSP: As indicated earlier, an additional node can be introduced, that encodes the division of the rendering space in two halves along a given plane. Again CSG operators are not natively supported and need to be converted before applying the algorithm.

Due to the nature of the BABEL DAG, the implementation for various functionality can easily be shared. This is very obvious if you consider the BSP node: once implemented it can trivially be extended to serve as optimization for the other two rendering algorithms.

4 The Implementation of BABEL

4.1 Solid and SolidNode

BABEL was implemented as a library in C++ using the VEGA Libraries [Tobl 95]. The complete interface of the data structure is provided by the so called **`Solid`** class. All implementation details are hidden from the user by this class. Each `Solid` represents a DAG (i.e. each `Solid` holds a pointer to a DAG) of **`SolidNode`**s, which provide the actual functionality (see figure 4.1). No matter what type of element is represented by such a DAG, the users handle for that object is a `Solid`: this is an intentionanl break of the C++ type system since the language type system was not considered powerful enough to support the structure we wanted. For this reason we had to implement our own type concept (this is described in the next section).

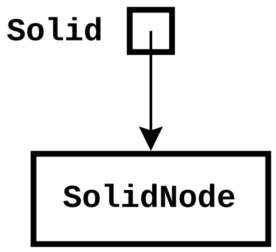

Figure 4.1: Solid and SolidNode

The main advantage of this approach is the fact, that one `SolidNode` can be shared by several `Solid`s, which reduces the amount of memory used (see figure 4.2). The classes `Solid` and `SolidNode` also implement a reference counting scheme for `SolidNode`s, so that the user does not have to bother with freeing the memory for `SolidNode`s when they are not used anymore (as soon as the reference counter of a `SolidNode` becomes zero, the `SolidNode` will automatically be freed, since no `Solid` points to it).

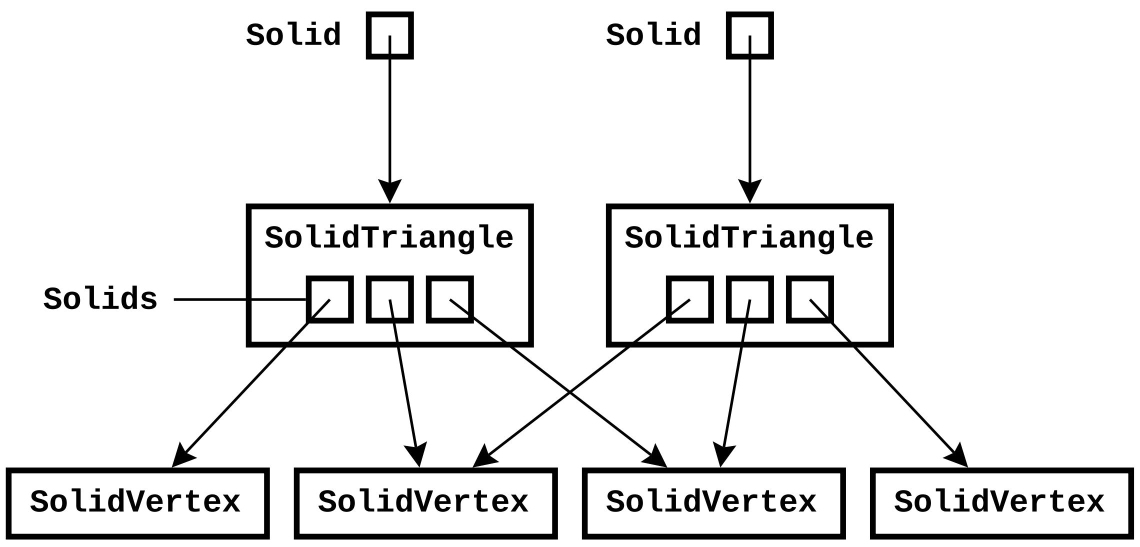

Figure 4.2: triangles sharing vertices

4.2 The Type Concept

BABEL uses an extended type concept to check the compatibility of two `Solid`s. Each `Solid` can have two different orthogonal types (which are totally independent):

implementation type: This type reflects the implementation of the `Solid`, and is defined by the class hierarchy. It is used to determine the exact type of a `Solid`. This type is not considered for geometric compatibility checks.

geometric type: This type defines the geometry of a `Solid`. It is used for compatibility checks in assignments and other statements. The geometric type does not depend on the implementation of the underlying node class.

Since C++ only supports a single type for each identifier at compile time, the use of two orthogonal types requires that the type checking is postponed until the program is actually executed. Therefore in BABEL type mismatches are only discovered at runtime.

Since the Solid class provides the only interface to the BABEL class hierarchy, it must have all abilities of every underlying class derived from `SolidNode`. Thus it has to mirror the functionality of all underlying classes. As the type and the provided functionality of a `Solid` cannot be checked at compile time, this check has to be done at runtime. If a function is called, that is not implemented for a specific `Solid`, a runtime error occurs.

4.3 The Class Hierarchy

Due to the separation of implementation type and geometric type, it was possible to base the design of the BABEL class hierarchy on implementation issues only. A small part of the BABEL class hierarchy can be seen in figure 4-3.

Within this class hierarchy, some classes which have very similar behavior are located at very different positions. As an example, SolidPolygon is derived from SolidNaryNode, whereas SolidTriangle is derived from SolidTernaryNode. But in a geometric sense, a SolidTriangle is a SolidPolygon. In order to support a hierarchy of geometric types, the implementation of the geometric type is based on the divisibility relation: each node class is assigned an integer tag. If the tag of class *A* is dividable by the tag of class *B* without remainder, then class *B* is a descendent of class *A* in the geometric type hierarchy.

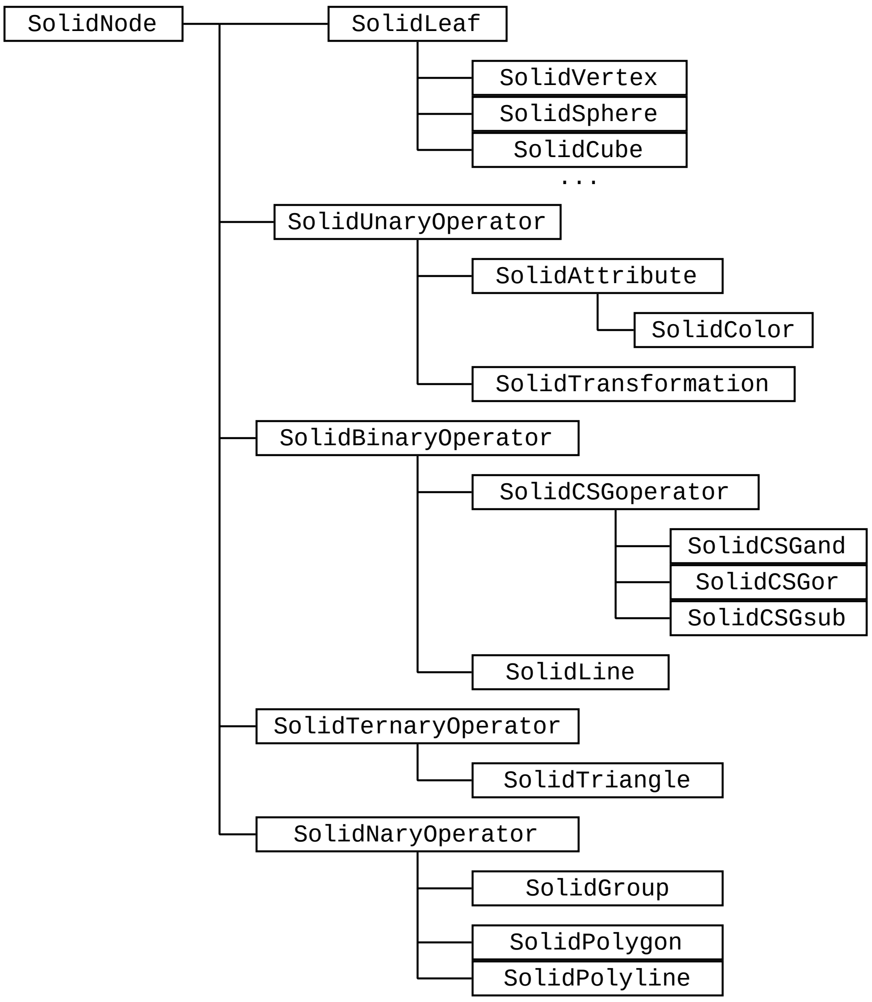

Figure 4-3: A part of the BABEL class hierarchy.

4.4 C++ as Modeling Language

Since C++ was used as implementation language for BABEL, and the complete interface to the functionality of the data structure is represented by one single class, operator overloading for this class makes it possible to use C++ as a high level modeling language for geometric objects and CSG models. This is comparable to the 2D-modeling of PASCAL/Graph [Bart 81]. C++ is a full-featured language without arbitrary restrictions as opposed to other Modeling languages like the language of the MIRA system [Magn 83] or SOL the modeling language of the RISS system [Gerv 88]. For this reason, various modeling tasks that need complex calculations to determine various object parameters can be easily accomplished. Another advantage of this strategy is, that future extensions of C++ will automatically be available for modeling purposes.

5. Conclusion and Future Development

The main functionality of BABEL together with a ray-tracer and an interface to the Silicon Graphics GL have been implemented. At our institute BABEL has been used in various projects for different purposes:

Modeling: A constraint based modeler was implemented using the BABEL data structures.

Virtual Environments: Various small VR test applications have been implemented using BABEL .

CSG to BREP conversion: A novel conversion algorithm for BABEL was implemented.

All these projects exhibited considerable savings in both, development time, and code size thanks to the high level of functionality provided by BABEL.

But BABEL is an open-ended project, and thus there are numerous ways to extend its functionality. Among these are:

Parametrized objects: Parameter nodes and parametrized attributes to support dynamic geometric objects.

Generalized transformations: including tapering, twisting and bending operators for simplification of modeling tasks

Space subdivisions: grid or octree nodes for fast access to geometric objects

Radiosity: attribute nodes specific to radiosity and a radiosity implementation that works on solids

References

[Appe 68] Appel, A.: “Some Techniques for Shading Machine Renderings of Solids”. *Proceedings of the Spring Joint Computer Conference*, pp. 37-45, 1968.

[Bart 81] Barth, W., Dirnberger, J., Purgathofer, W.: “The high-level Graphics Programming Language PASCAL/Graph”. *Proceedings of EUROGRAPHICS ’81* (ed. J. Encarnacao), North Holland Publ. Comp, pp. 151-164, September 1981.

[Buit 75] Bui-Tong, Phong: “Illumination for Computer Generated Pictures”. *Communications of the ACM* 18(6), pp. 311-317, June 1975.

[Catm 74] Catmull, E.: “A Subdivision Algorithm for Computer Display of Curved Surfaces”. Ph.D. Thesis, *Report UTEC CSc 74-133*, University of Utah, Salt Lake City, UT, December 1974.

[Fuch 80] Fuchs, H., Kedem, Z.M., Naylor, B.F.: “On Visible Surface Generation by A Priori Tree Structures”. *Proceedings of the SIGGRAPH ’80,* Computer Graphics 14(3), pp. 124-133, July 1980.

[Gerv 88] Gervautz, M., Purgathofer, W.: “RISS - Ein Entwicklungssystem zur Generierung realistischer Bilder” (RISS - A Development System for Genrating Realistic Images). *Visualiserungstechniken und Algorithmen* (ed. W. Barth), Informatik Fachberichte 182, pp. 61-79, September 1988.

[Magn 83] Magnenat-Thalmann, N., Thalmann, D.: “The use of high level graphical types in the MIRA animation system”, IEEE Computer Graphics an Applications 3(9), pp. 9-16, 1983.

[Roth 82] Roth, S.: “Ray Casting for Modeling Solids”. *Computer Graphics and Image Processing* 18(2), pp. 109-144, February 1982.

[Stro 91] Stroustrup, B.: The C++ Programming Language, 2nd edition, Addison Wesley, 1991.

[Tobl 95] Tobler, R.F., Löffelmann, H., Purgathofer, W.: “VEGA: Vienna Environment for Graphics Applications”. *Proceedings of the 3rd Winter School of Computer Graphics and Visualization*, pp. 323-238, Pilzen, February 1995.