Functional 3D Graphics in C++ — with an Object-Oriented, Multiple Dispatching Implementation

Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi
Integrated Media Platform Software
SunSoft, Inc.[1]
{gds,conal,ryeung,salim}@eng.sun.com

Abstract

Constructing interactive, animated 3D graphics applications has been notoriously difficult for well over twenty years. Even though significant advances in the state-of-the-art have been made, this situation persists. The system described here simplifies the programmatic construction of geometry in ways that we have not seen elsewhere, and does so within the framework of an accepted production language, C++. It has been our experience that the resulting programs are quite succinct and comprehensible, and execute efficiently. The programmer is presented with a simple, general interface that is both declarative and conforms to the functional programming paradigm. Pursuing a functional interface for developing interactive 3D applications is a novel concept that, in our experience, has been successful in providing a simple, powerful interface and a relatively straightforward implementation. The implementation of the system is highly object-oriented, relying heavily upon multiple dispatching. The system itself is extensible and adding new geometric primitives and operations is straightforward. Entirely new media types, such as sound and image, may be (and have been) added to the system.

Keywords: interactive, animated 3D graphics; geometry construction; functional programming; object-oriented graphics; multiple dispatching; multimedia; network distribution.

1 Introduction

This paper has two primary goals. The first is to describe the programmer's interface to the geometry-specific layer of the TBAG system, and communicate the level of expressiveness that can be achieved by using this interface. The second is to describe an efficient implementation of the interface that also allows for straightforward extensibility of the system.

TBAG is a general framework for creating interactive, time-varying integrated media applications, and is discussed in detail in [Elli94]. The geometry layer is a C++ interface for constructing and processing geometry "values". This interface conforms to the functional programming paradigm [Fiel88] and is very simple, yet general and powerful. It has been our experience that programs written with this interface are more succinct, comprehensible, and amenable to aggressive optimization techniques when compared with programs written using any other system we have seen.

Our interface has an efficient, highly object-oriented implementation. Furthermore, the system[2] is highly extensible in the sense that it is straightforward to add new geometric primitives and attributes as well as new operations to be performed on geometry values. The system may be, and has been, further extended to support additional media types, such as sound and image.

---

[1] 2550 Garcia Avenue, M/S MTV10-228, Mountain View, CA 94025 USA

[2] In this paper, when we refer to "the system", "the interface", or "the implementation", we mean the geometry-specific layer of the TBAG framework. When we are referring to TBAG itself, we will explicitly say "TBAG".

While TBAG provides sophisticated support for modeling interaction and time, for network distribution, and for the construction of collaborative applications, detailed discussion of these aspects is beyond the scope of this paper. The reader is encouraged to consult [Elli94] for more information on the aspects of TBAG not discussed here.

We believe that our approach of providing one programming paradigm (namely, the functional paradigm) to best meet the needs of the community using the interface, and another programming paradigm (the object-oriented paradigm) to best meet the needs of the extenders and implementers of the system is rare, and possibly unique, in the world of 3D graphics systems. In what is perhaps a somewhat heretical position to take within the object-oriented graphics community, we shall attempt to show why we feel that the functional approach is more appropriate for the user of the interface we provide, and how a functional interface can be provided while still retaining an efficient object-oriented approach for extension and implementation.

2 Related work

Easing the construction of interactive 3D applications has been a primary goal of a number of commercial and research systems. Traditional display list systems such as PHIGS [PHIG89], HOOPS [Wieg91], and Doré [Doré91] have existed for up to ten years and are available on a wide range of computers and operating systems. The design of these systems was largely driven by the capabilities of the graphics hardware that existed at the time. Thus, these systems are not known for their expressiveness or their simplicity.

More recent advances in object-oriented graphics systems have produced Mirage [Tarl92], Inventor [Stra92], GEO++ [Wiss90], and GROOP [Kove93]. These systems go a long way towards simplifying the conceptual model presented to the programmer. However, we don't believe that they afford as simple a representation of geometry as our system does, nor do they allow the same opportunities for aggressive optimization.

[Arya86] describes a functional approach to specifying discrete frames of an animation, but doesn't address the issue of constructing geometry. The Fugue system [Dann91], takes a functional approach to describing computer-generated music similar to our functional approach used for describing geometry, but invents its own language in the process. We know of no other system that provides a functional interface for geometry construction, let alone one that is targeted at building interactive applications using an accepted production language such as C++.

Inventor supports a multiple dispatching facility similar to the one described in Section 6.3. These facilities were developed independently and achieved similar results (although the facility presented here can be used in arbitrary places, unlike Inventor's). Mirage and DIVER [Goss93] provide for a declarative specification of lights, similar in result, if not in form, to that described in Section 7.1.

3 Programmer's Interface to the TBAG Geometry layer

The geometry layer is accessed through a functional programming interface that defines a number of abstract data types, constant values of these types, functions for creating values of these types ("construction functions"), and functions for combining values of these types into new values ("combination functions"). Note that there are no mechanisms provided to alter these values once they have been constructed. This principle of *immutability* results in many of the simplifications we have achieved, without sacrificing flexibility or efficiency. As mentioned in the introduction, we know of no other functional interface for constructing 3D geometry, let alone an interface in as commonly accepted a language as C++.

3.1 Types and their meanings

Values of type Geometry simply represent three-dimensional geometry. Some of the operations in our system allow Geometry values to be rendered into a frame buffer, to be intersected with a ray (in order to perform picking), to have their bounding box calculated, to have their shadow geometry created, etc.

When a value of type Attributer is applied to a Geometry value, a new Geometry value is created attributed with the Attributer. For example, if we attribute a cube geometry with the Attributer `edges_on`, the resulting geometry will be a solid cube with visible edges. The attribution operation is further discussed in Section 3.4.

Values of type Transform represent a geometrical transformation that can be applied to Geometry values to create a transformed geometry value. Unlike most systems, ours does not require Transforms to have a 4x4 matrix representation – in fact, the internal representation is completely hidden from the programmer. This allows our system to be extended to include new types of transformations, such as deformations, that cannot be represented as a 4x4 matrix.

In addition to Geometry, Attributer, and Transform, the system also provides types such as Color, Point, Vector, Axis, Quaternion, etc. From the point of view of the application programmer, types do not have a subclassing or inheritance relationship with other types. This is unlike most object-oriented graphics systems, and the reason for this will be explained in Section 4.3.

3.2 Constant values

Following are some declarations of constant values of various types. (TBAG is fully accessible through C++, and all of the examples in this paper are given in C++.)[3]:

```
extern Geometry&   cube;
extern Geometry&   sphere;
extern Geometry&   line_segment;
extern Color&      red;
extern Attributer& invisible;
extern Transform&  identity_trans;
extern Point&      origin;
extern Vector&     x_vector;
```

Note that C++ references (denoted by the ampersand) are used for most of the values. We find this preferable to using pointers. (Additionally, pointer values cannot be used with overloaded operators in C++ – a facility we make considerable use of.)

New values are created either by using a construction function or a combination function, as described below.

3.3 Construction Functions

Here are C++ declarations for a few construction functions:

```
// Color creation
extern Color& rgb_color(float r, float g, float b);
extern Color& hsv_color(float hue, float saturation, float value);

// Vector creation
extern Vector& rect_vector(float x, float y, float z);
extern Vector& spherical_vector(float theta, float phi, float rho);

// Axis creation
extern Axis& axis_along_vector(Vector& vec);

// Transform creation
extern Transform& xlt(float x, float y, float z);   // translation
extern Transform& scale(float x, float y, float z); // non-uniform scaling
extern Transform& rot(Axis& axis, float angle);     // rotation
extern Transform& rot_from_quat(Quaternion& quat);  // rotation
```

[3]The geometry interface is also completely accessible through standard C, with the exception of the overloaded operators `+` and `*`, for which other functions must be substituted.

```
// Attributer creation
extern Attributer& front_color(Color& col);    // create a front coloring Attributer
extern Attributer& model_xform(Transform& xf); // create a modeling transform Attributer

// Geometry creation
extern Geometry& stroke_text(char *text, Point& position,
                             Vector& horizontal_vec, Vector& vertical_vec);
extern Geometry& regular_polygon(int num_sides);
```

Note that there are often multiple ways to construct values of a given type. For instance, a Color may be created by specifying either RGB values or HSV values, a Vector may be created by specifying either Cartesian coordinates or spherical coordinates, etc. The internal representation of the values is completely hidden from the programmer.

3.4 Combination Functions

The geometry layer of TBAG derives much of its expressiveness from the way in which values may be combined to create new values. The two primary combination functions provide *aggregation* and *attribution*:

```
extern Geometry& operator+(Geometry& left, Geometry& right);
extern Geometry& operator*(Geometry& geo, Attributer& attr);
```

The "`+`" operator allows two Geometries to be combined into a new Geometry, and the "`*`" operator allows an Attributer to be applied to a Geometry and result in a new Geometry. Some examples:

```
cube + line_segment + cone      // combination of cube, line segment, and cone
cube * front_color(red)         // a red cube
cube * model_xform(xlt(1,1,1))  // a translated cube
cube * front_color(red) * model_xform(xlt(1,1,1)) // a translated red cube

// a translated, rotated red cube
cube * front_color(red) * model_xform(xlt(1,1,1))
                        * model_xform(rot(y_axis, pi/4)
```

Since attributing Geometry with a color or with a modeling transformation is such a common operation, we provide additional overloadings for "`*`" that allow a Color[4] or a Transform to be specified directly, rather than forcing them to be turned into Attributer's. This allows the last example to be rewritten as a more manageable:

```
cube * red * xlt(1,1,1) * rot(y_axis, pi/4)
```

Notice how these operations may be chained, and they compose as one would expect.

4 Design Principles Underlying the Interface

The above section described the interface to our system, without discussing any of the issues that influenced our design choices. Here we discuss these issues, and attempt to convey why we think the functional approach to specifying geometry is more concise, comprehensible, and amenable to optimization than other approaches we have seen, while retaining the useful flexibility that other systems possess.

4.1 Immutability of Values

One of the most important aspects of our system is that operations on values always construct new values, rather than altering existing values. Thus, `cube * red` does not change the color of `cube` to red, rather it creates a new geome-

[4]In fact, the overloading of `*` on applying a Color to a Geometry results in setting the front color, back color, line color, and text color.

try that represents a red cube. This is a key feature of the functional programming paradigm - values are constructed, not altered.

We say that our values are *immutable* in that, once constructed, there is no way to alter them. Thus, unlike most object-oriented graphics systems, these values are not “objects” in the traditional sense – they do not have mutable state nor do they have the corresponding methods to alter that mutable state. While at first this may seem like a debilitating restriction, we shall see in Section 5 why this is not the case. In fact, we shall see how this approach allows programs to be more succinct, more comprehensible, more amenable to both multithreading and aggressive optimization techniques, and provide straightforward support for network distribution. This approach may also introduce major concerns about efficiency. We shall show how our implementation overcomes these concerns.

4.2 Simplicity and Efficiency for Multithreaded Applications

Many multithreaded or re-entrant software systems become considerably more complex when they start having to deal with issues of mutual exclusion and locking around critical code sections in order to maintain a consistent shared state. In addition to the complexity of adding code to ensure consistency, such code also tends to decrease the level of concurrency achievable in such systems, since all other threads are locked out while one thread is in a critical section of code.

The functional approach of producing immutable values considerably simplifies the writing of multithreaded code that manipulate such values. Since the values are immutable, any number of threads can simultaneously access the same value without any fear of inconsistency or corruption. As a result, very high levels of concurrency may be achieved.

4.3 To Subtype or Not To Subtype?

We mentioned earlier that the programmer is not exposed to any subtyping or inheritance relationships between the types available in the system. Subtyping of an interface, when used properly, has the effect of extending the set of methods (protocol) available. From the point of view of the TBAG application programmer, subtyping is not useful, because all of our values are immutable (so they needn’t have methods that alter their properties), and because they are all capable of responding to the base protocol defined by the abstract type itself. Moreover, introducing subtypes encourages the introduction of functions and methods that can only operate on values of that subtype, thus making the interface less general. Our approach of not exposing subtyping increases the amount of information hiding or encapsulation in the system, bringing along the accompanying well-known benefits of information hiding.

Clearly, however, subtyping is a useful tool to be taken advantage of when it comes to extending a system. It is desirable to re-use as much interface and implementation as is possible. So how do we provide this advantage and still not expose subtyping to the programmer? The answer lies in our insistence on a strong separation between the needs of the application programmer and the needs of the system extender. We believe that with any successful system, while extensibility has to be straightforward, it is not as important as usability for the user who is not interested in extending the system. Thus, a system should not complicate the programmer’s model for the sake of the extender. A successful system will have a large community of users, the majority of whom do not extend the system, while a minority do extend the system.

To be consistent with this separation, we do not expose the programmer to any subtyping or inheritance relationships, but the system extender does have access to this information, and can do his or her own further subclassing when appropriate, thus re-using both interface and implementation.

4.4 Leveraging of Programming Language Facilities

As a design principle, we leave support for generally useful facilities to the host programming language wherever possible, thus distilling geometry support into as simple a form as possible, while allowing great flexibility and ease of expression. Consider creating a rectangular block with a ball on top of it, and having this geometry parameterized

by the color of the ball and the transformation that gets applied to the entire geometry. Rather than introducing a new mechanism into our system that allows for the parameterization of geometry, such as PHIGS structures or Inventor Node Kits, we simply use functions that may be defined in the host language (in this case, C++):

```
Geometry&
block_with_ball(Transform& xform, Color& color)
{
    return (cube * scale(1,4,1) * color     // appropriately colored cube
            + sphere * xlt(0,2,0) * yellow) // + yellow sphere
           * xform;                         // all transformed by xform
}
```

We now have a standard C++ function that represents the desired parameterized geometry.

4.5 Expressiveness of the Interface

As a measure of expressiveness, it's useful to compare how one would achieve a comparable result through a number of different systems. Consider creating a geometry that consists of a cube with edges turned on and a blue sphere, all of which are designated as unpickable. In TBAG, this would be expressed quite naturally and succinctly as:

```
(cube * edges_on + sphere * blue) * unpickable                    (1)
```

Note that in this expression, we specify *what* the geometry is, rather than specifying *how* to draw it.

In a traditional graphics system, like PHIGS, Hoops, or Doré, the Geometry in (1) would be expressed roughly as:

```
push pickability
set unpickable
push current edge flag
set edges on
execute cube
pop current edge flag
push current color
set color to blue
execute sphere
pop current color
pop pickability
```

Aside from being quite verbose, this latter expression forces the programmer to think in terms of a machine model of a graphics context and forces a description of *how* to draw the geometry.

The newer crop of object-oriented graphics systems, such as Inventor, GEO++, and GROOP, would express the Geometry (1) more like:

```
c = new Cube;
c->set_edges_on(True);

s= new Sphere;
s->set_color(Blue);

g = new Group;
g->add(c);               // add the cube to the group
g->add(s);               // add the sphere to the group
g->set_pickable(False);  // make the entire group unpickable
```

While this is considerably more understandable than the PHIGS style, it is still quite verbose and less comprehensible when compared with TBAG's expression of the geometry. It also leaves less room for the implementation to transparently perform geometry optimizations.

Why is TBAG so much more succinct and expressive than other systems? One aspect of the answer is that we don't have to worry about being able to edit geometry. All of the other cited systems have mechanisms that allow the edit-

ing of geometry, and this introduces considerable complexity into the entire interface. Another aspect of the answer is that our use of the syntactic features available in C++ make expression of Geometry quite natural.

5 Integration of the Geometry Layer into TBAG

The description of the geometry construction interface above focuses on creating static geometry values. A reader would be reasonably concerned about how this interface, with immutable values and no editing facilities, could be used to construct interactive, animated geometry. This section addresses that concern by describing the TBAG system itself. TBAG is a client of the Geometry-specific layer, and addresses issues that would be experienced by any system that built on top of the interface described above. TBAG provides mechanisms that let one express *time-varying* (i.e., animated and/or interactive) geometry in ways virtually identical to expressing static geometry values. We provide a very brief description of the facilities available in TBAG for doing this. The reader is encouraged to consult [Elli94] for more information.

TBAG is based on a very efficient local-propagation constraint solver, that, although it is by no means completely general, can solve an interesting and very useful subset of constraint problems. TBAG exposes the notion of a *constrainable*, which is an entity that may be involved in a constraint relationship. Constrainables are parameterized according to the type of the value they hold, and this type parameterization is expressed with C++ templates. There is a primitive Real-valued constrainable called `Time` that represents absolute time in seconds, and it may be used in constructing other constrainables. Some examples:

```
// cone rotating about the Y axis by 1 radian per second
Constrainable<Geometry&> geo = cone * rot(y_axis, Time);

// Pulsating block with ball whose hue is determined by Time, and whose
// scale is determined by the absolute value of the sine of Time. This makes
// use of the function we defined in Section 4.4.
Constrainable<Geometry&> geo =
   block_with_ball(uniform_scale(fabs(sin(Time))), hsv_color(Time, 1, 1))

// Constrainables may be parameterized as other types as well. Here, we construct
// a time-varying Point whose θ and ϕ are time-varying.
Constrainable<Point&> pt = spherical_point(Time, Time * 0.164, 1.0)
```

Note that many of the value construction functions that we saw in Section 3.3 are being used with constrainables as arguments. Even `block_with_ball`, a function defined by an application programmer, is being used with constrainables. How does this work in a statically typed language like C++? TBAG provides a tool that processes header files and creates overloaded versions of the functions in the header files. These overloaded versions are chosen by the C++ compiler if constrainables are provided as arguments to them. This tool allows sets of functions that were written without TBAG in mind to be used from TBAG, with TBAG constrainables.

In addition to being able to be expressed in terms of `Time`, constrainables may be put into constraint relationships with other constrainables. The system maintains the constraint, even when the contents of one of the constrainables involved in the relationship changes. This is accomplished via the “assert” function. Some examples:

```
Constrainable<Transform&> xf1;
Constrainable<Transform&> xf2;

// Assert that the combination of xf1 and xf2 is the identity transformation.
// This guarantees xf1 and xf2 to be inverses of each other.
assert(xf1 * xf2 == identity_trans);

// Create a geometry of two blocks_with_ball's, where one always has the
// inverse transform of the other.
Constrainable<Geometry&> geo = block_with_ball(xf1, red) +
                               block_with_ball(xf2, blue);

// Give xf1 an interesting trajectory. Because of its relationship to
```

```
// xf1, xf2 will automatically get the inverse trajectory, and thus both
// block_with_balls in geo will start moving.
assert(xf1 == rot(y_axis, Time) *
              uniform_scale(fabs(sin(Time)) *
              xlt(sin(Time), cos(Time * 3.0), sin(Time * -7.0))
```

Interaction is specified in much the same way as animation. The mouse, for instance, is abstracted as a time varying Point2D-valued constrainable. Asserting a constraint between the position of a geometry and the mouse constrainable causes the geometry to move when the mouse moves. Discrete events (for instance, a mouse button press) are modeled by retracting some constraints and asserting others. For instance, when a mouse button is released, the constraint tying the geometry to the mouse can be retracted, and a constraint tying the geometry to its current position will be asserted.

These techniques for expressing relationships between different program entities, and the relationship between different entities and time, frees the programmer from having to concern himself with any notion of flow control. He does not need to write an animation loop, nor does he need to receive timer events or mouse motion events, he simply expresses relationships for the system to maintain.

Hopefully these examples make the point that the features available for constructing static geometry are available for creating interactive, animated geometry as well. Because of our approach, we do not need the editing facilities that most graphics systems provide (these are the same editing facilities that make the systems considerably more complex). For example, rather than replacing a transform within a hierarchical structure to affect animation, we simply express the animation up front, and the system conceptually creates an entire new geometry value at each frame. (It's important to realize that this construction of an entire new geometry value at each frame is purely conceptual. The actual implementation is smarter than that.)

6 Implementation of the Interface

We now turn our attention to how the interface described in the preceding sections is implemented. Remember that none of what we discuss here is visible to the application programmer. The primary implementation concept for the geometry layer of TBAG, from which most of the other implementation concepts flow, is that of *rendering*. When we speak of rendering, we mean in it in the broadest possible sense. Rendering refers to the processing of a geometry value in order to achieve some result. As mentioned earlier, we use rendering to draw geometry values into a frame buffer, perform ray intersection, extract bounding boxes, etc.

There is a single function, declared as

```
extern void render_geometry_in_context(Geometry& geo, Context& ctx);
```

that embodies all the rendering possibilities we are faced with. The specific action taken by `render_geometry_in_context` is dependent upon the specific types of `geo` and `ctx`.

To understand how this single function can be used to control all rendering possibilities, one must understand how Geometry and Context values are created and represented in the system, and how the multiple dispatching facility operates. The following sections discuss these issues, as well as the memory management techniques that we use to make our approach feasible, some non-obvious techniques that make good use of available computational power, and some geometry-specific optimizations we are able to perform because of the approach we take.

6.1 Representation of Geometry values

Let's consider how geometry values are represented in the system. There is a class hierarchy extending from the `Geometry` abstract class that embodies all forms of geometry that may be created in the system. Here is a partial view of this hierarchy:

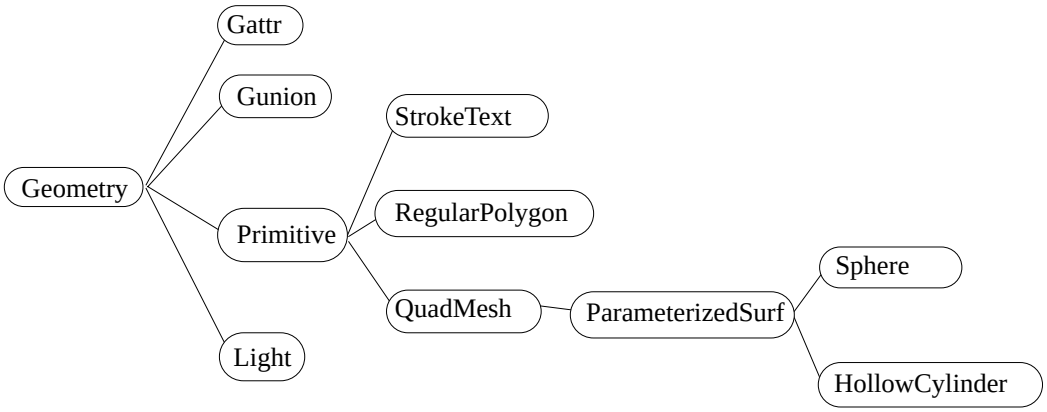

Figure 1. Partial class hierarchy for the `Geometry` type

Remember, as discussed in Section 4.3, that this hierarchy is not visible to those programming in TBAG. It is visible solely to implementers and extenders.

As an example of how a Geometry value gets created, consider the combinational operator "`+`" (discussed in Section 3.4) applied to two Geometries to create a new Geometry. `+` is defined as:

```
Geometry&
operator+(Geometry& left, Geometry& right)
{
    return *new Gunion(left, right);
}
```

and the `Gunion` subclass is defined as:

```
class Gunion : public Geometry {
  public:
    Gunion(Geometry& l, Geometry& r) : left(l), right(r) {}
    DECLARE_TAG(Geometry);

    Geometry& left;
    Geometry& right;
};

DEFINE_TAG(Gunion, Geometry);
```

There are a few items to take note of here:

- Generally, construction and combination functions return references to new instances of Geometry subclasses.
- The `DECLARE_TAG(Geometry)` and `DEFINE_TAG(Gunion, Geometry)` declarations are used to provide run-time type information. The reason for this will be described further in Section 6.3.2.
- Note that there are no methods defined on our `Gunion` class. All interesting functionality will be provided via methods selected by multiple dispatching.
- The `Gattr` subclass shown in the figure is similar to the `Gunion` subclass, but represents attribution, and is constructed via the `*` operator on a Geometry and an Attributer.

6.2 Contexts and their representation

A "context" is a program entity that accumulates state during the processing (traversal) of a Geometry value. The system contains a number of different context types, each specialized for a different type of "rendering". Here is a partial class hierarchy:

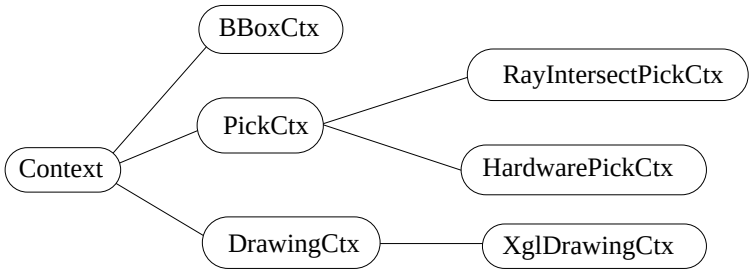

Figure 2. Partial class hierarchy for the `Context` type

A few notes:

- A BBoxCtx calculates the bounding box of the Geometry that's rendered into it.
- An XglDrawingCtx is a subclass of DrawingCtx, and uses SunSoft's XGL immediate mode 3D graphics library [XGL92] to do the rendering into a frame buffer. Other subclasses may be added to use alternate 3D rendering engines.
- There are a couple of different approaches for picking. One uses the underlying hardware support (accessible through XGL), while the other does ray-intersection based picking. In our experience, ray intersection picking is both faster and is capable of providing more complete results, including surface intersection point and normal.

The representation of a Context includes member data reflecting the current state of the Geometry traversal. For instance, the BBoxCtx is expressed roughly as:

```
class BBoxCtx : public Context {
  public:
    DECLARE_TAG(Context);

    ...
    void swallow_bounding_box(BoundingBox& bbox);
    BoundingBox *current_bbox;
};

DEFINE_TAG(BBoxCtx, Context);
```

The `swallow_bounding_box()` method updates `current_bbox` to include the specified bounding box. When traversal of a Geometry is complete, `current_bbox` holds the Geometry's bounding box.

6.3 The Multiple Dispatching Facility

6.3.1 Why we need multiple dispatching

The function `render_geometry_in_context` works by *multiply dispatching* off of its two arguments, `geo` and `ctx`. Multiple dispatching (sometimes referred to as *multiple polymorphism* or *multi-methods*) allows the code that actually gets selected to be executed (the "method") to be dependent upon the run-time types of two or more arguments of the "generic" function that was invoked. We borrow this terminology from CLOS, the CommonLisp Object System [Keen89]. The "generic" function being discussed here is `render_geometry_in_context`.

C++ supports only single dispatching, meaning that, given a virtual function `func`, the expression

```
obj->func(arg)
```

selects the particular `func` method to be executed based solely upon the run-time type of `obj`. The run-time type of `arg` can play no part in determining which `func` gets selected. In TBAG, `render_geometry_in_context` must choose the particular method to execute by taking into account the run-time types of both `geo` and `ctx`.

Multiple dispatching is essential for TBAG to be able to be a truly extensible system. An extender of the system needs to be able to add both new Geometry subclasses and add new ways in which Geometry gets rendered (i.e., add new Context subclasses). If we only supported single dispatching, then we would either have to define our Geometry abstract class to have a member function per type of rendering we wanted to support:

```
class Geometry {
  public:
    virtual void render_to_xgl(XglDrawingCtx& ctx) = 0;
    virtual void render_for_bbox(BBoxCtx& ctx) = 0;
       ...
};
```

Or we would have to define our Context abstract class to have a member function per type of geometry subclass we wanted to support:

```
class Context {
  public:
    virtual void render_gunion(Gunion& geo) = 0;
    virtual void render_stroke_text(StrokeText& geo) = 0;
    ...
};
```

Both of these alternatives are completely unacceptable for a truly extensible system, since, in C++, it is impossible to add methods to an existing abstract class without altering header files and recompiling the entire system. So, we have created an efficient multiple dispatching facility that we use throughout the system.

Contrast this approach to extensibility with those offered by most other graphics systems. Most other systems only allow the introduction of new primitives and (occasionally) attributes via the subclassing of existing primitives and attribues. It is generally not possible for a system extender to add in a new type of operation.

We use multiple dispatching for more than just the `render_geometry_in_context` generic function. For example, `compose_transforms` takes two Transforms and composes them, returning a new Transform whose representation is dependent upon the argument's run-time types. For example, a rotation composed with a rotation yields another rotation, while a rotation composed with a scale returns a Transform represented by a 3x3 matrix. `compose_transforms` determines which method to invoke by multiply dispatching off of its two arguments.

Additionally, we use multiple dispatching to control the processing of attributes in contexts. For instance, the processing of a color may be ignored in a ray-intersection context, but must be honored in an XGL rendering context.

6.3.2 Type tags

Since standard C++ doesn't currently expose any facility for portably retrieving run-time type information (RTTI), we need to provide our own facility for generating and accessing such information, in a portable fashion. The `DECLARE_TAG(AbstractClass)` and `DEFINE_TAG(Class, ParentClass)` macros used in conjunction with class definitions provide a per-class static "tag" that is unique across all tags declared to be part of `AbstractClass`. It also informs our run-time type facility that `ParentClass` is the most direct ancestor of the class in which the declaration is being made. In our implementation, the tag is simply a small integer. The static method `static_tag()` may be invoked on the class itself to retrieve the type tag of the class, while the method `tag()` may be used to retrieve the type tag of an object. Thus, to check if an object is of type `Gunion`, we could say:

```
Boolean is_gunion = (Gunion::static_tag() == obj->tag());
```

We need to supply `ParentClass` to the `DEFINE_TAG` macro in order for multiple dispatching to correctly reflect inheritance relationships amongst classes. As another example of the use of these macros, consider the declaration of the `Sphere` class, pictured in the class hierarchy in Figure 1.

```
class Sphere : public ParameterizedSurf {
  public:
```

```
    ...
    DECLARE_TAG(Geometry, ParameterizedSurf);
};

DEFINE_TAG(Sphere, ParameterizedSurf);
```

Note that our inheritance specification currently supports only single inheritance. We have yet to find a requirement for multiple inheritance in our system, and multiple inheritance is a facility whose usefulness has been called into question many times in the past.

It's regrettable that we need to specify inheritance and declare tags at all, since this information is already known to the C++ compiler. However, as mentioned before, there is no portable way to access this information, thus we need to duplicate the information. Even more unfortunately, the proposed C++ extensions for run-time type identification [Stro92] do not provide all of the information that we need to support multiple dispatching (specifically, information about inheritance relationships is not accessible).

6.3.3 Registering Methods

In order for multiple dispatching to do the work we want it to do, we need to register methods to be executed with particular subclasses. We do this via the `REGISTER_DOUBLE_DISPATCH_METHOD` macro. For instance, to register a method that extracts the bounding box out of a `Sphere`, we use:

```
REGISTER_DOUBLE_DISPATCH_METHOD(render_geometry_in_context,
                                BboxCtx, Sphere,
                                render_sphere_into_bbox_ctx);
```

A few comments:

- The first argument, `render_geometry_in_context()`, is the generic function that this method is being registered for. It has been declared as a generic function elsewhere.
- Method registration occurs at system initialization time, prior to any generic functions being invoked.
- `render_sphere_into_bbox_ctx` is most appropriately thought of as a member function of both the `BboxCtx` class and the `Sphere` class, and should have the same access privileges that actual member functions of these classes have. However, since C++ does not recognize these methods as member functions, we need to define our classes such that instance data is declared `public:`, thus allowing these methods access to that data. This is why `Gunion`'s `left` and `right` data members were declared public in Section 6.1.

The `render_sphere_into_bbox_ctx` function may be defined as:

```
static void
render_sphere_into_bbox_ctx(Sphere& geo, BboxCtx& ctx)
{
    // sphere's are always unit dimensioned.
    ctx.swallow_bounding_box(unit_bounding_box);
}
```

6.3.4 The Method Table and Inheritance

Conceptually, registering double dispatching methods on the `render_geometry_in_context()` generic function fills in a two-dimensional table of methods, with one axis representing the specific Geometry subtype and the other axis representing the specific Context subtype.[5] The method table may be sparsely populated, in that methods may

[5]In fact, our multiple dispatching facility is not restricted to just two dispatching arguments. A generic function that dispatched off of three arguments would conceptually have a 3D volume of methods.

not have been registered for every combination of argument types. It is the inheritance mechanism that determines which method to call if an empty table cell is reached by the dispatching facility. Consider the following simplified method table:

	A	B	C	D
X	6	4	2	
Y	5	3	1	
Z				

Here, with "P ← Q" meaning "P inherits from Q", we have Z ← Y ← X and D ← C ← B ← A. If we call a doubly dispatching function with arguments of type Y and of type C, conceptually the system will search for methods starting at #1 above, continuing in order to #6, stopping only when a method is found.

In practice, we use inheritance extensively. For instance, the method to render a `Gunion` object is the same regardless of the subclass of context used – simply render the left component into the supplied context, then render the right component into that context. Thus, we only define a `Gunion` rendering method on the abstract `Context` class, relying on inheritance to select this method for all other contexts.

As another example, consider the `Sphere` class. For rendering into an `XglDrawingCtx`, we inherit from the `ParameterizedSurf` class, which renders as a quadrilateral mesh. Since XGL doesn't support direct rendering of spheres, we can't do any better than rendering a `Sphere` as a parameterized surface. However, we do provide a method for rendering a `Sphere` into a `RayIntersectPickingCtx`, since performing ray intersection against a sphere is much, much faster than performing it against a general parameterized surface.

Our extensibility goals set forth in the introduction are easily met by the approach we have chosen to take. Adding both new geometry subclasses and new contexts (embodying new styles of rendering) have been shown to be straightforward. The inheritance features described here also make it straightforward to re-use existing code.

6.3.5 Implementing Inheritance Efficiently

The previous discussion about the method table and the method selection process may lead the reader to believe that our implementation's computational complexity in method selection is O(*mn*) with *m* being the number of arguments that we are dispatching off of (2, for double dispatching), and *n* representing the average length of the inheritance chain that needs to be traversed for each argument until a registered method is found or until the root of the inheritance chain is hit.

In our implementation, this is not the case, and we achieve O(*m*) performance, linear with the number of arguments being dispatched off of. We do this by eagerly filling in a sparse inheritance table during method selection. Specifically, if, prior to doing method selection, we have the following table:

	A	B	C	D
X	meth3	meth5		
Y				meth9
Z	meth7			

where the filled in cells represent registered methods. Now, if we invoke a generic function with arguments of types C and Y, the inheritance semantics will cause method selection to search the table and come up with `meth5`. However, during this initial lookup, we fill in all the empty cells we encounter, and end up with this table:

	A	B	C	D
X	meth3	meth5	*meth5*	
Y		*meth5*	*meth5*	meth9
Z	meth7			

with the italicized cells being newly filled in with `meth5`. Now, subsequent calls with arguments of type C and Y (or C and X, or B and Y) will not require any table searching.

We are able to perform this optimization because of a rather benign restriction we place on the system: No methods may be registered with a generic function after that generic function has been invoked for the first time. If we didn't have this restriction, the eager filling in of the sparse table would have incorrect semantics in the event that a new method was registered later on. We have yet to encounter a situation where this restriction was troublesome.

As a measure of the computational efficiency of our approach, we examined the assembly code resulting from compiling our multiple dispatching facility. We found that, after the first invocation, a double dispatch requires just 53 SPARC instructions to find the appropriate method. If deemed necessary, this number can be halved by making changes that would result in a small decrease of flexibility.

6.4 Memory Management Techniques

As discussed in Section 5, one constructs an interactive animation in TBAG by using constrainables as arguments to functions. Consider displaying a cube that is being scaled by the absolute value of the sine of time and whose hue is varying with time:

```
cube * hsv_color(Time, 1, 1) * uniform_scale(fabs(sin(Time)))
```

If this is displayed starting at time t = 10.0, at a rate of 20 frames per second, the first five displayed frames would consist of the following geometries, respectively:

```
cube * hsv_color(10.00, 1, 1) * uniform_scale(fabs(sin(10.00)))
cube * hsv_color(10.05, 1, 1) * uniform_scale(fabs(sin(10.05)))
cube * hsv_color(10.10, 1, 1) * uniform_scale(fabs(sin(10.10)))
cube * hsv_color(10.15, 1, 1) * uniform_scale(fabs(sin(10.15)))
cube * hsv_color(10.20, 1, 1) * uniform_scale(fabs(sin(10.20)))
```

Given the earlier description of how geometry values are constructed, it should be clear that each of these expressions creates four new values – one Transform, one Color, and two `Gattr` Geometries (via the `*` operator). If values are allocated and constructed every frame, one would expect that an excessive amount of garbage will be created, and memory usage and allocation would become a major problem.

We avoid this potentially fatal problem by recognizing that, because of our functional, immutable approach, every value that is constructed during a frame is only needed for the duration of that frame. After the frame is displayed, the values are no longer needed. To exploit this fact, all TBAG objects derive from a base class that redefines the C++ `new` operator to, when appropriate, allocate from a "transient store". Thus, while in "transient mode", all TBAG objects are allocated sequentially from a small number of large chunks of memory. At the end of a frame, all of the created objects are collected simply by resetting a few pointers in the transient store. The next frame writes over the same chunks of memory.

6.5 Techniques for Computational Efficiency

A reader may understandably get the impression that evaluating the expression

```
cube * xlt(1,1,1) * rot(y_axis, pi/3);
```
(2)

creates a Geometry value by translating each point in the vertex lists representing the cube by (1,1,1), then rotating each point around the *Y*-axis by $\pi/3$. Such an approach would be computationally prohibitive for non-trivial geometries, would not take advantage of the property that transforms compose with each other, and would not utilize any on-board graphics acceleration hardware that can deal with transforms directly.

Our system does not take this obvious approach. Rather, the expression in (2) above effectively evaluates into the binary tree shown in Figure 3. When, for example, this representation is rendered into an XGL context, the translation

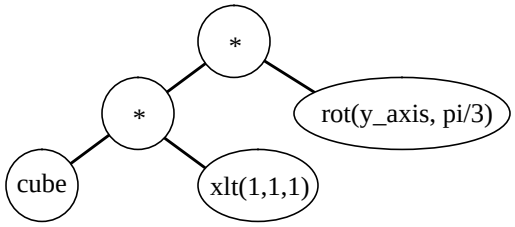

Figure 3. Representation of Expression (2)

and rotation transformations are composed and sent into the graphics hardware, making the best use of the available computational facilities. If this expression is part of a graphics animation, and the rotation amount is changing each frame, then a new binary tree will be built each frame. It turns out that, because of its simplicity, building such a tree is very inexpensive and, as explained earlier, all memory used in building it will be reclaimed the following frame.

6.6 Static Geometry Optimization

Consider rendering the expression `cube * xlt(1,1,1) + cube * rot(y_axis, pi/4)` to an XGL context. Such an operation involves setting the modeling transform to the translation, drawing a cube, setting the modeling transform to the rotation, and drawing another cube. This does not make particularly good use of the underlying graphics hardware, which is geared more for accepting large, batched geometric primitives. However, if we explicitly ask for this geometry to be "optimized", it will be converted into a single "multi-simple polygon", a batched set of individual polygons. This is done by "pushing through" the transforms into the primitives themselves, and coalescing the resultant primitives into a single primitive. Not only does this make optimal use of the underlying graphics hardware, but it is also an optimization that a programmer would probably be unwilling to perform manually.

Note that this optimization is feasible only because we do not allow editing of a geometry value. If we did, then the original geometry would need to be retained, and any changes to it would somehow need to be forwarded to the optimized version.

In our current system, geometry optimization has to be invoked explicitly by the programmer, by applying the function `optimize_geometry` to a Geometry value. In the future, we intend to have this be implicit. Specifically, when a geometry becomes constant (e.g., a user was manipulating the geometry, but has since released it), the system will automatically invoke optimization. This is similar to the dynamic compilation that the Self system [Cham91] performs after a function has been executed a suitable number of times, thus suggesting that the future usage pattern warrants accepting the cost of compiling that function, and will be a fairly straightforward addition to our system.

6.7 Implementation Statistics

The TBAG system is the result of approximately three calendar years and ten person-years of design and implementation. TBAG's geometry layer is a robust, mature, consistent, and full-featured system. It consists of 34 different Geometry subclasses, 12 Transform subclasses, 14 Context subclasses, over 100 Attributer subclasses, and over 15 other abstract data types. The geometry layer consists of about 20,000 lines of C++ code, all of which is consistent with the principles laid out in this paper. The system performs very well on both medium- and high-end graphics hardware – particularly on the SPARCstation 10ZX. Graphics performance is comparable to programs hand-coded in

XGL. As an example, a comparison was done between rendering 500 spheres each tessellated into 50 quadrilaterals. On a SPARCstation 10ZX, the geometry specified in terms of TBAG operations rendered within 3% of the speed of the geometry specified more directly in terms of the underlying graphics library.

7 Additional Features

The geometry layer of the TBAG system has some additional interesting features that we have not seen in other systems. This section describes some of them.

7.1 Embedded Lights

Lights are first-class Geometry values in our system. Thus, they can be embedded directly in geometry expressions and have attributes applied to them in the same way that any other geometry can. In particular, they can be transformed along with the geometry. As an example, consider the expression:

```
(cube + positional_light(yellow)) * xlt(1,1,1)
```

Here, `positional_light(yellow)` creates a positional light at the origin (all positional lights are initially at the origin and may be transformed from there) that emits yellow light. It is combined with a cube, and the entire compound geometry is translated to (1,1,1). Thus, we have a cube with a yellow light in the center of it, all at (1,1,1).

Most other interactive 3D graphics systems either force lights to be specified in world coordinates, thus they cannot be part of the model itself (PHIGS takes this approach), or, as is the case with Inventor, lights only affect the objects that come after them in the display hierarchy. This latter approach forces geometry description to be order-dependent (thus eliminating opportunities for optimizations based on primitive re-ordering), and hence non-declarative.

7.2 Embedded Shadow Planes

We also provide a “shadow plane” geometry that, like a light, may be trivially combined with other geometries. A shadow plane should be thought of as an invisible plane initially in the *XY* plane at $z = 0$. It may be transformed from that position into other positions. For each light source in the scene, every other geometry casts a shadows onto each shadow plane. On a SPARCstation 10ZX, this is all accomplished at interactive frame rates for reasonably complex scenes.

Figure 4 shows a scene composed of 9 geometries combined together using the “+” operator described in Section 3.4: a shadow plane, two directional lights, two geometries that provide a visual representation of each light, and four canonical shapes. Note the shadows cast onto the shadow plane by each pair of shape and light.

7.3 Support for Interaction

TBAG supports the specification of geometry values that have their own behavior under interaction. *Manipulators* are packaged up as Attributers, and may be used to attribute geometry (just as colors and transforms attribute geometry) so that the geometry behaves in a way dictated by the manipulator when it is interacted with. Most other systems decouple the behavior of an object under interaction from the specification of the appearance of that object, making for less declarative applications. Inventor is a notable exception, as it allows interaction nodes to be part of its graphs.

7.4 Integration of Sound

We have integrated audio support into our system in a manner entirely consistent with our geometry system, and sound and geometry can easily be combined together to create intriguing applications. For instance, the expression:

```
chirping_sound * frequency(1.2) * amplitude(0.8)
```

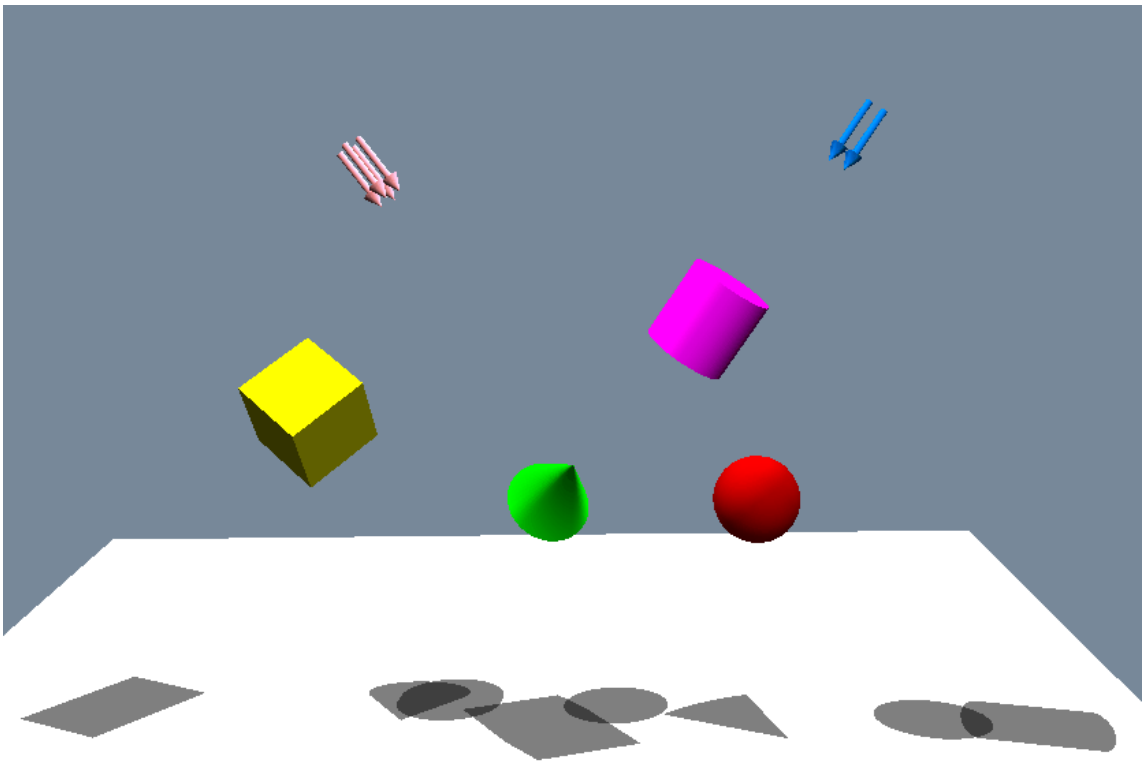

Figure 4. Scene with Lights and Shadow Planes

creates a Sound value representing a bird chirping (`chirping_sound` should be thought of as a constant Sound, in the same manner that `cube` is a constant Geometry), attributed with attributes stating that the sound should play back 20% faster than normal, and with an amplitude of 80% of normal.

A Sound may be incorporated into a Geometry via the `sound_at_origin` construction function. `sound_at_origin` takes a Sound and returns a Geometry representing the specified sound at the 3D origin. This geometry can then be transformed to other places in the usual manner. The rendering of the resultant geometry takes into account the current "Listening Transform" (akin to the 3D graphics notion of viewing transform), and plays the sound in such a way that the listener experiences an approximation of the sound actually coming from the specified location.

We have used this facility to create an application that presents a 3D landscape of about 30 different sounds, each associated with a 3D icon (see Figure 5). The user has the ability to navigate a pair of microphones through this landscape, hearing the sounds that are in the proximity of the microphones through stereo headphones.

We intend to integrate 2D imaging (and thus video, as time-varying 2D images), into our system in a similar fashion. This will allow arbitrary combinations of imaging and graphics media to be easily expressed. For example, texture mapping a wall in a 3D scene with an image produced from compositing a source image with the rendering of another 3D scene will be straightforward to express in our system. (Seeing that it can execute quickly on today's graphics and imaging hardware is another story!)

7.5 Network Distribution

TBAG provides a very simple way to create distributed, collaborative applications. Examples of such applications include:

- *Collaborative design*: three designers are all viewing the same geometry on workstations around the country. Modifications that any of the three make are witnessed by the other two, as they are happening.

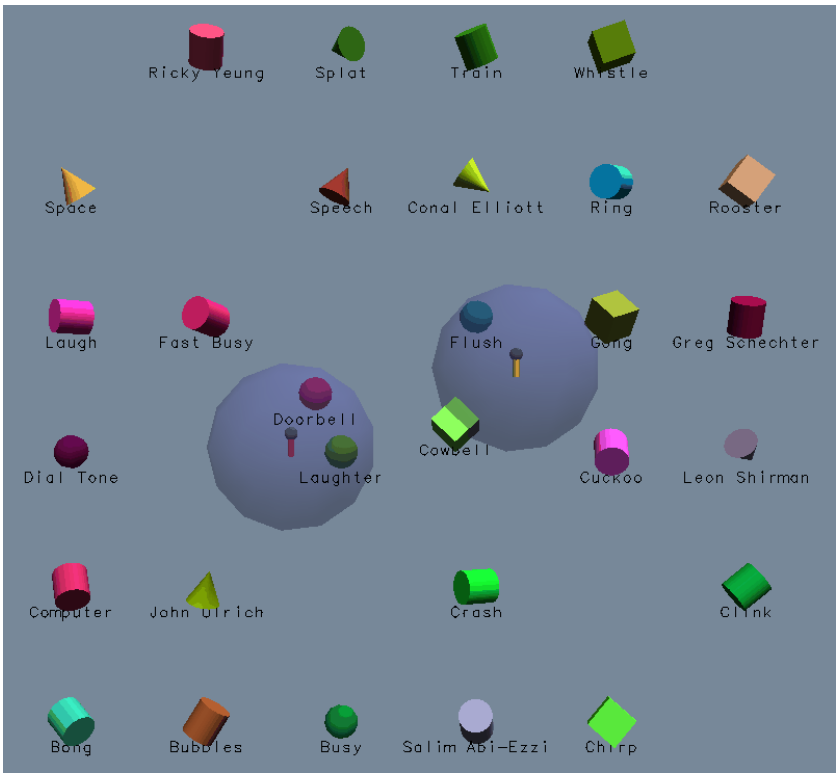

Figure 5. The SoundScape Application

- *Remote tutoring*: a professor is teaching a undergraduate physics course and presents an electronic illustration of spring forces in action. Students are watching this experiment live from their dorm rooms, their homes, or around the world may interact with the experiment on their computers, to get a better understanding of the physics of springs.

Not only are the distribution and collaboration aspects of these types of applications simple to construct in TBAG, but they also execute quite efficiently, using very little network bandwidth.

[Elli94] discusses the implementation strategy for achieving this description of distribution and collaboration. However, to make it work, a requirement is placed on the Geometry-specific layer. Namely, all “values” described in Section 3 need to be able to “print” themselves into a machine-independent representation that can be interpreted on a remote machine. The interpretation of this machine-independent representation will create a new value whose behavior is indistinguishable from the old value.

The immutability of values makes providing the machine-independent printed representation somewhat easier. This is because the printed representation of a value is essentially an encoding of the expression that was evaluated to construct the value in the first place. If the values were mutable, then the expression that led to the value’s creation would not necessarily reflect its current state.

8 Conclusion

In this paper, we’ve described the geometry layer of the TBAG system. We have presented a simple and general functional interface for programming 3D graphics that is fully accessible from C++. We’ve also described the design principles that underlie the interface, and discussed the advantages gained in succinctness and comprehensibility by presenting a functional interface to the programmer. In addition, we’ve shown how the interface is used from within the TBAG framework to create interactive, animated programs.

The presented interface poses some interesting challenges to the implementation. We have described our current implementation and explained how it supports the interface in both a time- and space-efficient manner, and how we provide general extensibility. In particular, a multiple dispatching facility that accounts for the inherent extensibility and much of the computational efficiency of our approach was explained in depth.

We have also discussed a number of interesting features of our system including embedded lights and shadow planes, support for interaction, integration with sound, and aspects of TBAG's transparent support for network distribution.

9 Acknowledgments

Thanks go to Leon Shirman, Srikanth Subramaniam, Ajay Sreekanth, and Tom Meyer for their help developing, commenting on, and using the TBAG system. Doug Gehringer, Randy Pausch, Russell Pflughaupt, and Matthias Wloka commented on earlier drafts of this paper. We'd also like to thank Matt Peréz for his support of this project.

References

[Arya86] Kavi Arya. A Functional Approach to Animation. In *Computer Graphics Forum*, 5(4):297-311, December 1986.

[Cham91] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Lisp and Symbolic Computation*, 4, 243-281, 1991.

[Dann91] Roger B. Dannenberg, Christopher Lee Fraley, and Peter Velikonja. Fugue: A Functional Language for Sound Synthesis. In *IEEE Computer*, July 1991.

[Doré91] *Doré Programmer's Guide, Release 5.0,* Kubota Pacific Computer, Inc., 1991

[Elli94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. To appear in *Computer Graphics (SIGGRAPH '94 Proceedings)*, volume 28(2), July 1994.

[Fiel88] A.J. Field and P.G. Harrison, *Functional Programming,* Addison-Wesley, Reading, Mass., 1988.

[Goss93] Rich Gossweiler, Chris Long, Shuichi Koga, and Randy Pausch. DIVER: A Distributed Virtual Environment Research Platform. In *IEEE Symposium on Research Frontiers in Virtual Reality.* October, 1993.

[Keen89] Sonya E. Keene, *Object-Oriented Programming in Common Lisp*, Addison Wesley, 1989.

[Kove93] Larry Koved and Wayne L. Wooten. GROOP: An object-oriented toolkit for animated 3D graphics. In *OOPSLA '93 Proceedings*, October 1993.

[PHIG89] Programmer's Hierarchical Interactive Graphics System (PHIGS). International Standard ISO/IEC 9592.

[Stra92] Paul S. Strauss and Rikk Carey. An Object-Oriented 3D Graphics Toolkit. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26(2), July 1992.

[Stro92] Bjarne Stroustrup and Dmitry Lenkov, *Run-Time Type Identification for C++ (Revised yet again).* ANSI document x3j16/92-0121, WG21/N0198, 1992.

[Tarl92] Mark A. Tarlton and P. Nong Tarlton. A framework for dynamic visual applications. In *1992 Symposium on Interactive 3D Graphics*, pages 161-164, 1992.

[Wieg91] Garry Wiegand and Bob Covey, *HOOPS Reference Manual, Version 3.0*, Ithaca Software, 1991.

[Wiss90] Peter Wisskirchen. *Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems.* Springer-Verlag, Berlin Heidelberg, 1990.

[XGL92] *Solaris XGL 3.0 Reference Manual*. Sun Microsystems, Inc., 1992.