TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications

Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi
SunSoft, Inc. *

Abstract

We present a paradigm and toolkit for rapid prototyping of interactive, animated 3D graphics programs. The paradigm has its roots in declarative programming, emphasizing immutable values, first class functions, and relations, applying these concepts to a broad range of types, including points, vectors, planes, colors, transforms, geometry, and sound. The narrow role of modifiable state in this paradigm allows applications to be run in a collaborative setting (multi-user and multi-computer) without modification.

CR Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism; I.3.6 [**Computer Graphics**]: Methodology and Techniques; D.1.1 [**Programming Techniques**] Applicative (Functional) Programming; D.2.m [**Software Engineering**] Miscellaneous *Rapid Prototyping*; G.1.7 [**Mathematics of Computing**] Ordinary Differential Equations.

Additional Keywords and Phrases: Local Propagation Constraints

1 Introduction

TBAG is a paradigm and toolkit for rapid prototyping of interactive, animated 3D graphics programs, based on two broadly applied design principles: graphical ADTs (abstract data types), and explicit functions of time. TBAG attempts to make parameterized geometric models as easy to express as mathematical formulas, by providing a set of *high level graphical ADTs* and functions and operators for constructing graphical values. These types include points, vectors, planes, colors, transforms, geometry, and sound. Values of these types are immutable, ensuring that different uses of a value, even ones occurring in parallel, cannot interfere with each other.

A single type of entity, the *constrainable*, represents modeling animation parameters of all types, user interaction, and even entire animations. Constrainables explicitly represent functions of time, to be sampled automatically by TBAG, thus relieving application programmers from involvement with frame generation and input device "motion events". Functions and operators that have been defined to work on basic types, including TBAG's high level graphical types, are automatically overloaded to work on constrainables over those types, producing new constrainables. The result is an almost invisible syntax for constructing interactive animations.

Other features of TBAG include lights, shadows, and even sound integrated with geometry in conceptually consistent manner. Also, velocity, gravitational and spring forces, etc., can be specified with ADTs (ordinary differential equations), which are formulated as equality expressions involving the constrainable derivatives. Finally, TBAG supports networked distribution transparently.

While other computer graphics researchers have done much good work to extend the might of numerical constraint solvers, our own work is complementary. It shows how to take a simple and efficient constraint solver, apply it uniformly to a multitude of types (including very high level types), make it support a continuous time model, and provide an almost invisible syntactic interface to it. Future work could merge these two research paths.

In this paper, we present details of TBAG's design, and how we resolved the implementation challenges that resulted. We then discuss TBAG's support for transparent and efficient distribution. Next support for derivatives and ODEs are described. A collection of sample applications is then presented. Finally, we make comparisons with related work.

2 The TBAG Programmer's Model

In this section, we describe TBAG's conceptual model and the C++ interface that embodies it. We also discuss how TBAG applications are developed. Implementation issues are discussed in later sections.

2.1 Graphical Data Types

TBAG programs perform geometry-specific operations by constructing and manipulating instances of high-level graphical data types. In the design of these types, TBAG leaves generally useful facilities to the host programming language wherever possible. Thus, type-specific support is distilled into as simple a form as possible, while allowing great flexibility and ease of expression. For instance, in TBAG,

- *Definition of reusable geometry* is handled by programming language support for definition of constants of arbitrary types.

*2550 Garcia Avenue, M/S MTV10-228, Mountain View, CA 94025 USA.
phone:415-336-{3086,6950,1791,2141}.
email: {conal,gds,ryeung,salim}@eng.sun.com.

- *Parameterized reusable geometry* is handled by definition and invocation of functions that produce geometry. The resulting customization power is not found in currently available graphics packages. Moreover, parameters may be of any type, including the geometry type itself.
- *Attributes and their scoping* are supported by functions that take the involved geometry and attributes as parameters (using a convenient infix notation), and yield a new geometry value.

To see how this approach works in practice, suppose one wants to construct several "block with ball" geometry values, consisting of a rectangular block of a specified color and a yellow sphere on top of it, together transformed according to some specified modeling transformation. In order to create several of these assemblies instead of just one, the programmer would define a C++ function[1]:

```
Geometry
block_with_ball(Transform& xform, Color& color)
{ return
  ( unit_cube * scale(1,4,1) * color +
    unit_sphere * xlt(0,2,0) * yellow ) *
  xform;
}
```

Comments:

- The types Geometry, Transform, and Color are predefined abstract data types. TBAG supplies these types and others, such as Point, Vector, and Axis, together with constants and functions for constructing values of those types.[2]
- The constants `unit_cube` and `unit_sphere`, each of type Geometry, are unit-dimensioned shapes, centered at the origin. The constant `yellow` is of type Color.
- The functions `scale` and `xlt` take three real numbers and construct a modeling transform, i.e., a value of the abstract type Transform, to scale or translate, respectively, the specified amounts along the *x*-, *y*-, and *z*-axes. There are also functions to create rotation and uniform scaling transforms, compositions of transforms, and transforms defined by matrices.
- The overloaded operator "`*`", when applied to a geometry value and a transform value, yields a transformed geometry value. In this example, the first application yields a non-uniformly scaled cube. The same operator is also overloaded to take a geometry value and a color and produce a colored geometry. In fact, both of these operations are special cases of a general attribution operation, which is based on an abstract type Attributer. (Other, less commonly used, attributers do not have additional overloadings, and instead are constructed via functions that indicate the specific attributes involved. For instance, `edge_color` is a function from colors to attributers, so `cube * edge_color(red)` evaluates to a cube with red edges.)
- The overloaded operator "+", when applied to two geometry values, yields a geometry value that is the union of the two given geometry values. (Hence any operation that can be applied to geometry values can be applied to unions formed by "+".) The final use of "`*`" transforms this geometric union.

1. C++ note: "`T&`" is the type of references to values of type `T`.

2. Each of these types is defined by a C++ abstract class, and some built-in subclasses, which makes TBAG conveniently extensible at this level of "predefined" types, constants, and functions. For ease of extensibility, the Geometry class's main method is actually a multi-method, choosing the actual code that is executed based upon the run-time types of more than one argument. See [20] for more details.

It is important to keep in mind that, conceptually, every step involved in this definition creates a new value rather than side-effecting an existing one. Different uses of the same value, therefore, cannot interfere with each other, and so TBAG programmers need not be concerned with the order or frequency of evaluation of their definitions.

Geometry values support several operations, including rendering, picking, and bounding-box determination. Most TBAG programmers, however, need not even be aware of these operations, which are invoked automatically by *viewer* objects, and automatically supported by geometry values built up with the primitive geometry values, functions and operators provided in TBAG.

2.2 Constrainables

The style of expressions used for creating the static, non-interactive values described above may also be used to create dynamic, interactive *"constrainables"*. A constrainable represents a conceptually continuous flow of values, out of which the application (or the system) can retrieve a value corresponding to a specific time using the type-parameterized function `value_at`:[3]

```
template<class T>
T value_at(Constrainable<T> cbl, Real time)
```

A TBAG application creates constrainables that embody portions of the desired animation and interaction. Animated, interactive geometry is represented by a Geometry-valued constrainable, i.e., a value of type `Constrainable<Geometry&>`. A TBAG *viewer* object simply invokes `value_at` repeatedly on a Geometry-valued constrainable with the current time, and renders the result to a window on the screen, thus producing animation.

The remainder of this section describes how applications introduce constrainables, how constrainables get their values, and how constrainables are put into relationships with other constrainables.

2.2.1 Primitive Constrainables

Constrainables are built up compositionally, out of a few types of primitive constrainables, representing time and both physical and virtual input devices. The Real-valued constrainable `Time`, when asked for its value at a given time (supplied by a viewer), returns that time.

As an example of an input device, consider a window-system slider:

```
Valuator<Real>& hue_slider =
  real_slider("Hue", 0, 2 * pi, 0);
```

The function `real_slider` creates a labeled slider on the screen with a minimum, maximum, and initial value. The resulting object contains a public instance variable, `value`, which is a Real-valued constrainable representing the setting of the on-screen slider at all points in time. (Without infinite buffering, only part of the past of an input device may be accurately queried.) The TBAG system ensures that the position of the on-screen slider and the `value` constrainable always remain consistent.

3. C++ note: this kind of declaration can be interpreted as an infinite family of function declarations, with `T` ranging over all types.

Similarly, a constrainable representing the time-varying position of a 2D mouse relative to an on-screen window can be accessed as:

```
Valuator<Point2D&>& mouse_val =
  mouse_constrainable_from_x_window(x_window_id);
```

Now, `mouse_val.value` is a Point2D-valued constrainable representing the position of the mouse relative to the specified window at all points in time. Primitive constrainables representing input devices are used to drive interactive applications.

2.2.2 Compound Constrainables

The TBAG system includes a tool that processes C and C++ header files and produces C++ overloaded functions that will accept constrainables as arguments and return properly typed constrainables. Thus, any function that was defined to operate on a type of value may be used to operate on constrainables over that type. For instance, evaluating the expression `sin(Time)` produces a Real-valued constrainable c such that, `value_at(`c`,`t`)`, returns `sin(`t`)`. The standard file `math.h` was processed through the overloading tool to provide the overloading for the `sin` function.

These expressions may be nested to an arbitrary depth. For instance:

```
xlt(sin(Time), cos(Time * 2.0), 1.0)
```

produces a Transform-valued constrainable that, at time t, evaluates to `xlt(sin(`t`), cos(2`t`), 1.0)`.

Time-varying, interactive geometry is created in a similar fashion. Consider the `block_with_ball` function defined in Section 2.1. Once its header file has been processed through the overloading tool, it may be used with constrainables as arguments:

```
block_with_ball(uniform_scale(fabs(sin(Time))),
                hsv_color(hue_slider.value, 1, 1))
```

The result is a Geometry-valued constrainable representing a block-with-ball that is scaling according to the sine of time, and whose hue is determined by the setting of the slider built in Section 2.2.1.

The importance of the overloading tool cannot be overemphasized. With it, we have automatically turned hundreds of functions that were written without regard to constrainables into functions that can take constrainables as arguments and produce new constrainables (including all of TBAG's geometry-related functions).

Since TBAG allows time-varying and interactive values to be expressed directly, the programmer need not be concerned with many issues relating to flow of control. In most other systems, the application programmer needs to take one of two approaches:

- Explicitly poll for changes on input devices and explicitly update animation parameters.
- More commonly, use an event-driven system, and would register interest in input and timer events, with the registered callback procedures taking responsibility for correctly updating the relevant parameters.

In TBAG, neither approach is necessary, since the desired behavior is explicitly encoded into the constrainables. (In Section 2.3 we shall see that inherently discrete input events, such as a mouse button click, are treated differently.)

2.2.3 Establishing Relationships Among Constrainables

A constrainable may be related to other constrainables by setting up *constraints* among them (hence the name "constrainable"). The following code sets up a geometry consisting of a red and a blue block-with-ball, each positioned according to a Transform-valued constrainable:

```
Constrainable<Transform&> xform1, xform2;
Constrainable<Geometry&> scene =
  block_with_ball(xform1, red) +
  block_with_ball(xform2, blue);
```

At this point, the two blocks in `scene` have no connection with each other, so they may be manipulated independently (via `xform1` and `xform2`). Next, we can make these two transforms interdependent, with their composition being the identity transform:

```
assert(xform1 * xform2 == identity_trans);
```

The == operator returns a *constraint*, and `assert` tells TBAG to enforce the specified constraint. The `assert` function returns an *assertion*, which may later be retracted.

Once this constraint is asserted, changes to one of the transform constrainables result in changes to the other. For instance, moving one causes the other to be moved in the opposite direction; shrinking one causes the other to grow; and rotating one causes the other to perform the opposite rotation.

In TBAG, constraints may be asserted on constrainables of any type, and the system determines the appropriate values the constrainables should take on in order to satisfy the required constraints. From the programmer's point of view, constraints are continuously maintained among continuously time-varying values.

2.2.4 Multidirectionality

In the above example, we do not say that `xform1` is dependent upon `xform2`, or vice versa. Rather, we say that `xform1` and `xform2` are interdependent. This symmetry allows us to exploit *multidirectionality* — a powerful feature of the constraint engine that underlies TBAG. Thus, `xform1` can be altered and we would expect `xform2` to vary accordingly in order to maintain the relationship, or `xform2` can be altered and `xform1` remains consistent. One way to visualize this situation is to consider the code in Section 2.2.3 as creating the undirected constraint graph in Figure 1. [1]

When the application assigns a value to `xform1`:

```
// assign time-varying rotation:
Assertion *ast = assert(xform1 == rot(y_axis, Time));
```

the constraint engine produces the directed constraint solution graph shown in Figure 2, causing `xform2` to be dependent upon `xform1`. Since the transform composition operation can run in any of three directions (given any two inputs, calculate the third as output), the application can retract the above assertion and assign a value to `xform2`:

1. "Constant constrainables" are shown as dashed circles.

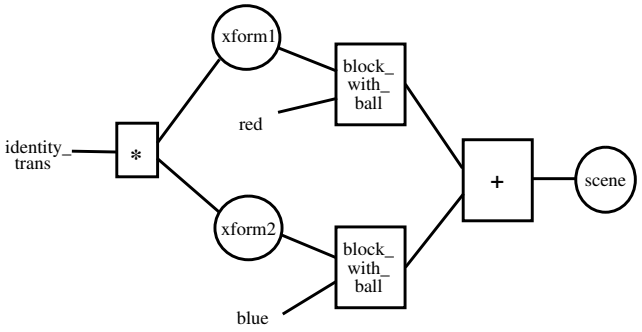

Figure 1. An Undirected Constraint Graph

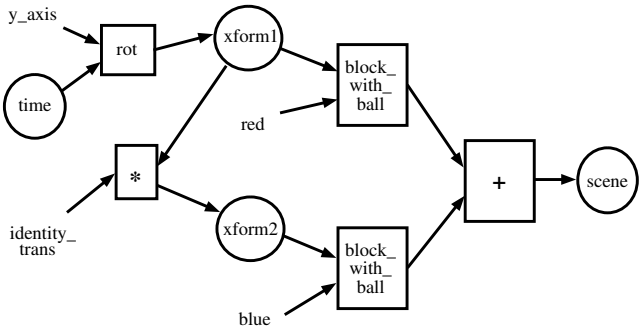

Figure 2. Figure 1 with `xform1` determining `xform2`

```
retract(ast);
// assign time-varying uniform scale
ast = assert(xform2 == uniform_scale(Time));
```

thus producing the new graph shown in Figure 2.

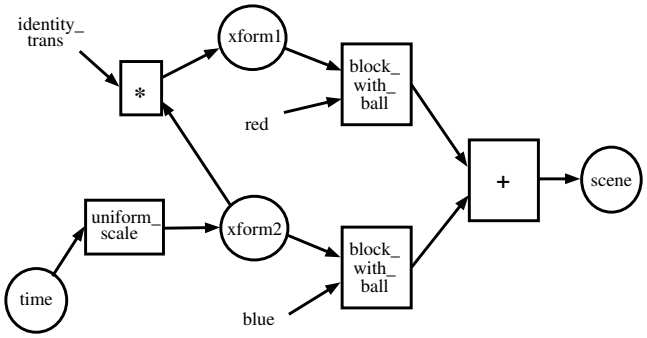

Figure 3. Figure 1 with `xform2` determining `xform1`

Note that if `ast` had not been retracted prior to the second assertion, the application would have been notified of a conflict (because the composition operation would have three inputs and no outputs).

2.2.5 Constraint Generality vs. Efficiency

When considering the use of constraints in an interactive context, it is vital to find a good compromise between generality and efficiency. Ideally, one would like extreme generality, but that choice leads to computationally intractable (and even undecidable) problems. On the other hand, a constraint system that sacrifices too much power for the sake of efficiency will not satisfy programmers' needs for expressiveness.

One approach for dealing with this problem would be to extend the constraint system to deal with new situations as they arise. In general, this is not a satisfactory approach since: it may require in-depth knowledge of the constraint system to solve new problems; *ad hoc* extensions to the constraint system may result in decreased efficiency for the system in general; and the designer of a system cannot possibly predict all the situations an application programmer might encounter.

TBAG has adopted a pragmatic approach to dealing with this problem. Rather than forcing the constraint system to be extended for each new situation encountered, TBAG makes it straightforward and convenient for an application programmer to create and register application-specific functions and constraint-solution methods for those functions. These methods are invoked by the underlying constraint system at the appropriate time.

2.3 General Approach for Interaction

As noted above, interaction that is conceptually continuous is encoded directly into constrainables, and thus the application doesn't need to deal with tracking events from conceptually continuous devices. Examples of conceptually continuous interaction include window system slider motion, mouse motion, the turning of a dial, and six degree-of-freedom head tracking.

There are, however, other interactions that are fundamentally discrete (event based). Examples include button presses and menu choices. TBAG applications generally deal with such discrete input events by retracting some existing assertions and asserting new constraints. For instance, in a slight variation of the "opposite block-with-ball" example above, clicking on a block causes that block's position to be related to the mouse position (and, because of the relationship between transforms, the other block moves in the opposite direction). Releasing the mouse button removes the relationship between the mouse and the block. As in this example, discrete events tend to alter the topology of the constraint graph.

2.3.1 Manipulators

In the above scenario, the application does not need to watch for "motion events", but it does need to watch for discrete events, like a mouse button press or release. To help applications respond to discrete events, TBAG packages up common interaction paradigms into "manipulators" that may be used to attribute Geometry values, thus making the attributed Geometry obey a particular interaction paradigm. For instance, while the geometry given by

```
block_with_ball(xform1, red)
```

does not know how to respond to the mouse, the result of

```
block_with_ball(xform1, red) *
xform_manipulator(xform1)
```

does. The function `xform_manipulator` takes a Transform-valued constrainable and returns an Attributer that, when attached to a geometry, causes the transform (value of the constrainable) to become constrained to the mouse when the geometry is picked, and releases the transform from the mouse when the geometry is released.

2.4 Constraint Classes

Some TBAG programs introduce hundreds or thousands of constrainables and similar numbers of constraints. To make TBAG scalable, there needs to be a way to manage this complexity. Fortunately, as pointed out in [3], there is an elegant idiom for dealing with this complexity problem, combining the use of classes with constraints. TBAG applications make frequent use of these *constraint classes*, which define, as instance data, constrainables representing the object's constrainable properties. Thus, instantiating such a class automatically creates a new set of constrainables. Class initialization code (the C++ constructor method), assert constraints that relate properties of these constrainables. Other constraints relating the newly created object to the outside world are added outside of the class initialization code. Furthermore, class inheritance results in "constraint inheritance" for free. The reason is that if a class B derives from a class A, then initialization of objects of class B will include their initialization as A objects, including the assertion of A's constraints. As an example, Figure 4 shows a hierarchy of geometric constraint classes. Note how the subclasses inherit the constraints of the parent classes.

```
class GeometricObject {
  public:
    Constrainable<Geometry&> geometry;
    // No constraints
};

class XformedGeometricObject: public
                      GeometricObject {
  public:
    Constrainable<Geometry&> base_geom;
    Constrainable<Transform&> model_trans;
    // Constraints:
    XformedGeometricObject() {
      assert(geometry == base_geom *
                         model_trans); }
};

class AffineGeometricObject : public
                     XformedGeometricObject
{
  public:
    Constrainable<Point&>      position;
    Constrainable<Quaternion&> orientation;
    Constrainable<Vector&>     scale_vector;
    Constrainable<Vector&>     shear_vector;
    // Constraints:
    AffineGeometricObject();{
      assert(model_trans ==
          scale_by_vector(scale_vector) *
          shear_by_vector(shear_vector) *
          orient_by_quat(orientation) *
          translate_to_position(position)); }
};
```

Figure 4. Some geometric object classes

2.5 Lights

In TBAG, lights are simply geometry values. Unlike PHIGS [17] and immediate-mode graphics libraries [27,6], lights may be embedded in scene geometry, subject to that geometry's modeling transforms. This ability facilitates creation of richly articulated geometric assemblies that contain light sources that move appropriately. Lights influence the appearance of all geometric objects in the same scene. (In contrast, programmers of IRIS Inventor must worry about order of traversal, since lights only affect objects "later" than them in traversal order. On the other hand, in TBAG, there is no way to limit the scope of influence of a light.) The implementation price paid for this convenience is that two traversals of the geometry are made. In the first, lights are accumulated and transformed to world coordinates (WC), and non-light geometry is ignored. In the second traversal, the previously ignored geometry is rendered as lit by the accumulated WC lights.

There are four geometry-creating functions corresponding to light types (ambient, directional, positional, and spot lights). Each has useful defaults for all arguments.

As an example, consider the following C++ function:

```
Geometry&
directional_light_shaped_as_cone(
     Color& col, Vector& light_direction)
{ return
  (cone * col + directional_light(col)) *
  align_vectors(light_direction, z_vector);
}
```

This function takes a color and a direction and produces a cone of the specified color emitting a light of that color, all aligned with the specified direction. (The utility function `align_vectors` takes two Vector values and produces a transform that maps the first vector to the second.)

TBAG is not the only system that treats lights as first-class objects (e.g., Mirage [24] and DIVER [10] also do), but it is the only system that we know of that allows such succinct integration of lights with scene geometry.

2.6 Shadows

The interface for shadows is based on the notion of a *shadow plane*, which is an invisible, oriented plane that catches shadows. If a scene contains shadow planes, then by default, every pair of non-ambient light and visible geometry casts a shadow onto every shadow plane. With the current renderer, shadows are simply black, semi-transparent, singular transformations of the visible geometry.

The programming interface for shadows consists of a new Geometry constant, `shadow_plane`, and a few new constant attributers.

- The constant `shadow_plane` is a shadow plane situated in the *XZ* plane, oriented so that shadows are cast on the positive *Y* side. To create shadow planes with other locations and orientations, one applies transforms to this canonical shadow plane.
- The geometric attributes `light_creates_shadow` and `light_creates_no_shadow` control whether lights in their scope create shadows, with the former being the default.
- The geometric attributes `cast_shadow` and `cast_no_shadow` control whether visible geometry in their scope cast shadows onto shadow planes.

Figures 7, 8, and 13 show examples of shadow planes in use.

2.7 Sound

TBAG makes it easy to add synchronized sound to geometric animations. Sound support is based on the following principles:

- Sound is an abstract data type, like Geometry, with primitives, attributes, and combination.
- Interactive, animated sounds (sounds that are changing based on time and other parameters) are supported as Sound-valued constrainables.
- Synchronization between geometry and sound works simply by relating geometry constrainables and sound constrainables to some of the same parameter constrainables.

To illustrate this last point, the following code fragment constructs a rotatable cube whose rotation angle is determined by a slider. The rotating cube emits the sound of a flute with the pitch modified by the angle's rate of change.

```
Constrainable<Real> angle =
  real_slider("Rotate cube:", 0, 2*pi, 0).value;
Constrainable<Geometry&> scene =
  cube * rot(y_axis,angle) +
  sound_at_origin(flute*frequency(derivative(angle)));
```

Some comments:

- `flute` is a constant Sound of a flute playing with a canonical pitch and volume.
- The function `frequency` takes a real number and produces a sound attributer that when applied to a sound value multiplicatively modifies the frequency of that sound. Other sound attributers include phase and amplitude.
- The function `sound_at_origin` takes a sound value and produces a geometry value, which is the sound argument embedded into geometric space at the origin. As such it has no visual appearance but may be heard. Such geometries may then be transformed elsewhere in 3D, and renderers may choose to perform audio spatialization on the sound to impart a sense of position to the sound. The current TBAG renderer performs very basic audio spatialization.

3 Implementation

The programmer's model presented above has an efficient implementation that makes a number of non-obvious design choices. This section presents that implementation and discusses some of the design choices.

3.1 Behaviors

TBAG was designed to provide a fundamentally *continuous*, rather than discrete, treatment of naturally continuous phenomena such as time and motion. The basis for implementing TBAG's continuous approach efficiently is the *behavior*, a type-parameterized family of immutable data types that represent first-class functions of time. The notion of behaviors applies pervasively, for all types and at all levels, from individual parameters to entire animations. For instance, a time-varying angle is represented as a Real-valued behavior, the position and orientation of a geometric component as a transform behavior, and an entire animated scene, as well as each of its geometric components, as geometry behaviors. Behaviors are purely an implementation device, being constructed invisibly during constraint solution, as will be described below in Section 3.2.

Behaviors support a *sampling* operation, which produces a value for a given time *t*. For instance, consider an animation, which is represented via a geometry behavior. An application or user creates any number of viewer objects, connected to the same geometry behavior. Each of these viewers iteratively samples the geometry behavior according to some criterion, such as maintaining a desired frame rate. In addition, each viewer also has a (viewing) transform behavior, which it samples for the same sequence of times. Each geometry and viewing transform sample pair is rendered to produce a frame of animation. Various viewers of a single geometry behavior may sample it completely independently, e.g., with different frame rates. No interference among these viewers is possible, due to the immutability of TBAG values.

In addition to sampling, behaviors also support differentiation, integration, and fairly general systems of first-order and higher-order ODEs, all over an extensible collection of types. For instance, the derivative of a point behavior is a vector behavior. (See Section 5.) Differentiation is done analytically when possible, and numerically otherwise (when differentiating input devices or functions whose analytic derivatives are not known). Integration and ODE solution is done analytically when of a very simple form, and otherwise using a standard efficient and accurate numerical technique (fourth order Runge-Kutta with adaptive step-size determination).

3.2 Continuous Constraints

Our goal with respect to constraints has been to explore easy expression and application to high level (non-numeric) types, rather than powerful numerical constraint solution techniques. TBAG currently uses the SkyBlue constraint satisfaction algorithm [19], which is a descendent of DeltaBlue [14]. Constraints may be specified on arbitrary types of (immutable) values, and may be given different strengths. This efficient, incremental algorithm ensures that a globally optimal subset of specified constraints are satisfied. As explained below, efficiency of TBAG's constraint maintenance is also significantly enhanced by an unconventional use of the underlying constraint satisfaction algorithm, which allows it to be invoked relatively infrequently.

TBAG's design imposes unusual requirements for the constraint system.

- In order to support a fundamentally continuous model of time and interaction, TBAG's constraint system must ensure that the constraints involving continuously changing values are continuously maintained.
- Efficient prediction of constrained values must be possible (and is in fact the typical case).[1]
- The instantaneous rate of change of a constrained value must be available, in order to support derivative constraints. As will be described in Section 5, derivative constraints allow natural expression of behaviors governed by derivatives, integrals, and differential equations.

None of these requirements hold in typical constraint-based systems, because such systems are usually data-driven, i.e., input values are allowed to change discretely, after which the system makes other discrete value changes to resatisfy the constraints. One could satisfy the first requirement by changing the constraint system

from data-driven to demand-driven, propagating value demands backwards, rather than propagating value changes forwards. This change would be based on the observation that while, conceptually, values (such as the geometry of a scene) are continuously changing, these values will be sampled discretely (e.g., by viewers during automatic frame generation). The second and third requirement, however, would still not hold in this hypothetical constraint system.

TBAG's answer to the three requirements above is to make the values contained in constrainables and manipulated by the discrete constraint system be *behaviors* instead of base-level values. The constraint engine responds to assertions and retractions by constructing new behaviors for constrainables (typically corresponding to user interactions and other significant events), and propagates such (discrete) changes forward to cause related constrainables to take on new behaviors. (Interestingly, the underlying data-driven constraint system's type-genericity allowed it to be used in this unusual way without modification.) The first requirement is satisfied because the desired constraints hold among the related behaviors, and hence among all corresponding samples. The second requirement follows from the fact that behaviors support arbitrary time-sampling. (Actually, the prediction is only correct as long as the constrainable has its current behavior, which is a useful approximation of prediction of constrained values.) The third requirement follows because derivative is a well-defined concept on behaviors.

A significant performance benefit of this approach is that it allows the constraint engine to run infrequently. The constraint system is only invoked when an application changes underlying relationships in the system. Because the constraint system relates behaviors, its results tend to be valid for a significant amount of time. Section 2.3 illustrated how this characteristic is exploited during user interaction.

Keep in mind that the TBAG programmer is unaware of the fact that constrainables actually contain behaviors and are modified discretely, and instead think of constrainables informally as containing basic values and being modified continuously.

3.3 Efficiency Techniques

Any high level paradigm naturally raises concerns about performance. Although TBAG is quite high level, it is nonetheless satisfactorily efficient. The following subsections describe our efficiency techniques. Note that all of these optimizations are possible precisely because of the immutable nature of graphical data and behaviors.

1. To appreciate the motivation for prediction, consider that a frame of animation is computed starting at some earlier time t_C, but will be rendered at some later time t_R, so the system should compute what the geometry and viewing transform will be at time t_R, rather than what they are at time t_C. This discrepancy is especially noticeable with user interaction, because both the geometry and the viewing transform may be dependent on continuous input devices (such as a 2D or 6D mouse). In particular, when the viewing transform is derived from a head tracker, users are extremely sensitive to any lag in response to head motion. This "interaction lag" is avoided by the fact that all behaviors, including input device behaviors, are capable of doing prediction. Of course neither input values nor the time t_R can be predicted with certainty. However, the alternative of not doing prediction is equivalent to predicting that the input values will remain constant between t_C and t_R.

3.3.1 Efficient Memory Allocation

Conceptually, for any given viewer object, a scene's geometry and viewing transform are computed from scratch at each frame. (Much per-frame construction can often be avoided, as described in Section 3.3.2 below.) Because the nature and size of the geometry representations are not known *a priori* and because they outlive the function calls that construct them, they must be allocated dynamically (not on the stack) each frame. In order to reduce program complexity and error, TBAG frees the programmer from the need to explicitly free up the allocated storage for abstract values.

At first glance, this approach would seem to require prohibitively expensive use of garbage collection. Fortunately, this is not the case. Due to the side-effect-free nature of behavior execution, any memory allocated during the computation and rendering of a single frame becomes unreferenced after the frame's rendering is complete. Thus, TBAG viewers use a very efficient form of allocation, which we refer to as *transient allocation*. While in "transient mode" (e.g., during frame generation), abstract TBAG values are allocated sequentially from a small number of large chunks of memory. Leaving transient mode (e.g., at the end of a frame) causes these chunks to be made available for future reuse, at the cost of resetting a few pointers.

3.3.2 Dynamic Constant Folding

Geometric objects tend to contain many more potential degrees of freedom (represented as constrainables) than are being varied at any one time. Thus, most behaviors are potentially varying but currently constant. A naive implementation of a continuous, demand-driven constraint system, however, would result in repeated evaluation of all behaviors, even the constant ones. Such an implementation would preclude scalability. The TBAG approach of immutable behaviors contained in mutable constrainables allows for a simple solution to this problem. When a compound behavior is constructed, the argument behaviors are checked. If they are all constant behaviors, a new constant behavior is created (for which the result value is computed lazily). Since compound behaviors may be arbitrarily nested, this constant behavior creation may trickle up to larger and larger portions of the constructed behavior. We call this process "dynamic constant folding." Note that explicit invalidation is never necessary, since it happens automatically when the constraint engine recomputes behaviors.

3.3.3 Explicit Geometry Optimization

The TBAG function `optimize_geometry` takes a geometry value *g* and returns a geometry value *g'* that responds to geometry operations indistinguishably from *g* but more efficiently. As usual, there is a trade-off to consider: more time is needed to construct the optimized geometry value *g'* than to simply use *g* instead, so the optimization is only worthwhile if the result will be used repeatedly.

Different optimizations are done for different operations. For instance, to support rendering, flattening of geometric hierarchy is done to (a) reduce run-time stacking and composition of transforms, and (b) present large graphics primitives (e.g., triangle strips and quadrilateral meshes) to the rendering engine. The actual optimization for each operation is postponed until the operation is first used.

Although not done currently, geometry optimization could apply automatically, e.g., in conjunction with dynamic constant folding.

4 Distribution and Collaboration

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

- *Collaborative design*: three designers are all viewing the same geometry on computers around the country. Modifications that any of the three make are witnessed by all, as they are happening.
- *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 homes, in the classroom, or distributed throughout the state, and 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.

4.1 Writing Collaborative Applications in TBAG

Distribution and collaboration are facilitated by the addition of a few functions to the TBAG programming interface. One group of functions allow constrainables and assertions to be mapped into machine-independent identifiers that can be passed to different processes on different machines. These are known as "externalization" functions. The other group of functions "internalizes" the externalized identifiers back into C++ constrainables and assertions. The programmer thinks of the externalize/internalize sequence of calls as providing access to an existing constrainable or assertion in a separate process, thus allowing constraints to be asserted on and values to be retrieved from those remote constrainables.

These functions tend to be used as follows. A TBAG application has created a constrainable that produces an interesting animated, interactive geometry. To publish its existence, the application externalizes the constrainable, and communicates the machine-independent identifier to other TBAG applications (perhaps via electronic mail, RPC, ToolTalk™, CORBA™, etc). After receiving the externalized version of the constrainable, these other TBAG applications internalize the identifier and are left with what appears to be a standard C++ constrainable. The application may then do what it pleases with that constrainable. Any changes made to the constrainable are reflected in all processes that are accessing that constrainable.

4.2 An Efficient Implementation of Distribution

The above description can be implemented in a number of ways. An obvious approach would be to have process B's internalized constrainable identifier simply contain a reference to the original constrainable on process A. While this approach could work, it would suffer considerable performance penalties. Specifically, each time process B does a `value_at` on the constrainable, a network round trip will need to be made to A to retrieve the value, incurring an unacceptable amount of latency and network usage.

Our approach to implementing distribution avoids these inefficiencies by using replication and local execution where possible. Whenever a process creates a constrainable, all other involved TBAG processes create "clones" of that constrainable. Then, whenever a constraint is asserted in any process, all related TBAG processes are informed of that assertion and perform it themselves locally. Similarly, when a constraint is retracted, related processes perform the retraction locally. Thus, each process has a semantically identical copy of the entire constraint network. As explained in Section 3.2, because interaction and animation are encoded directly in constrainables, applications tend to make assertions and retractions relatively infrequently. Therefore, the expense of keeping the constraint graphs consistent is small.

Not all constraint graphs can be entirely replicated via cloning on all machines involved in a collaboration. Consider the collaborative scenario in Figure 5. This figure represents the programmer's

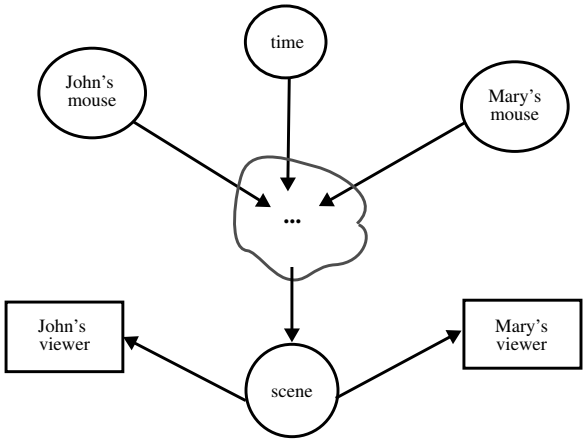

Figure 5. Programmer's view of collaborative application

view of a collaborative application, with the nebulous blob in the center representing a single constraint graph that, from the two mice and time, is determining a geometry to display on both John and Mary's computers. The naive implementation discussed earlier would produce each "frame" of geometry and send it to each display, thus using considerable bandwidth.

However, what our implementation actually does is reflected in Figure 6, where the constraint graph inside the nebulous blob has been replicated onto both machines, and the only information that needs to flow from one machine to the other are mouse positions.

This approach to efficient distribution via replication depends on the ability to distribute constraint assertions and retractions between processes. Such a requirement forces the "values" that constrainables hold to be able to produce a machine-independent representation of themselves. Thus, if an application executes `assert(geom == unit_cube * red)`, the implementation needs to be able to communicate a machine-independent representation of `unit_cube * red` to other processes. Our system achieves this by requiring every implementation subclass of an abstract data type to know how to "print" its instances. This "printed" representation can later be evaluated on another machine to produce a new value that behaves identically to the original one.

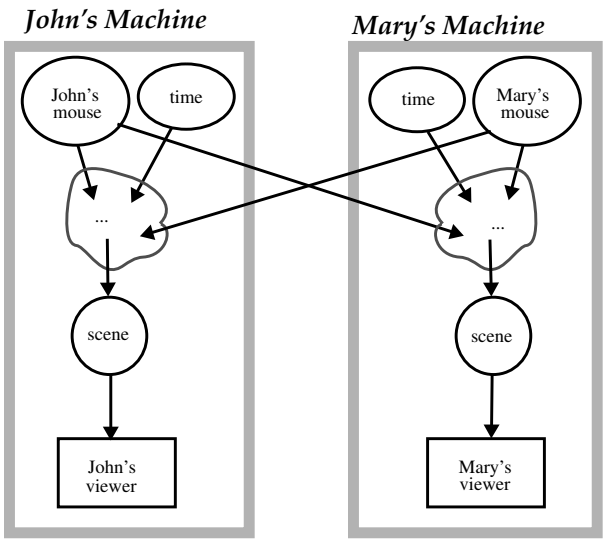

Figure 6. Implementation's view of collaborative application

There is one troublesome aspect to the implementation approach for distribution described above. Every constrainable gets replicated on every participating process, and every assertion and retraction is performed on every participating process, whether or not the process is interested in the constrainables involved. For larger-scale distributed applications with many participants entering and leaving at will, there needs to be a better way, such a "lazy cloning" of constraint graphs.

5 Derivatives, Integrals, and ODEs

The combination of derivatives and constraints is a very powerful specification tool for animation and interaction. In fact, our desire to support derivative constraints originally motivated TBAG's behavior-centered internal design. As a simple example, the following lines create two Real-valued slider widgets and constrain the value property of one to be equal to the derivative of the value property of the other.

```
Valuator<Real>& val_slider =
  real_slider("Value", -10, 10, 0);
Valuator<Real>& rate_slider =
  real_slider("Rate", -10, 10, 0);

assert( derivative(val_slider.value) ==
        rate_slider.value )
```

The derivative of a constrainable *v* is another constrainable *v'* with the constraint that the derivative of *v* (considered as a function of time) is equal to *v'*. Consequently, after this constraint is asserted, when a user grabs and moves the value slider, the rate slider continuously moves to reflect the instantaneous rate of change of the value. When the value slider is not moving, the rate slider stays at zero. Also, however, the user may grab and move the rate slider and see the value slider gradually increase and decrease its value at a rate controlled by the rate slider. The constraint system automatically chooses between differentiation (in the first case) and integration (in the second).

To illustrate how derivatives can be used in interaction, one can relate the value slider to a modeling transform of a geometric object being viewed, as follows.

```
assert(my_xform == rot(y_axis, val_slider.value));
```

The result is that a user may control the model's rotation angle via the value slider, or its rotational velocity via the rate slider.

Higher-order derivatives are easy to express, simply as repeated applications of `derivative`, or as a chain of derivative constraints, as follows.

```
Valuator<Real> accel_slider =
  real_slider("Acceleration", -10, 10, 0);
assert(derivative(rate_slider.value) ==
       accel_slider.value));
```

After performing this assertion, all three sliders affect each other and the model's rotation appropriately.

As illustrated above, two techniques to solve derivative constraints are *differentiation* and *integration*. Another technique that turns out to be very powerful is the extraction and solution of systems of ordinary differential equations (ODEs). The need to solve ODE systems arises when there are *cyclic* sets of derivative constraints — a situation that occurs quite frequently in practice.

As an example of derivative constraints requiring ODE solution, consider Newton's law of linear motion, $f = m\,a$. This physical law could be encapsulated as a class that introduces the notions of force and mass:

```
class ParticleObject : public AffineGeometricObject {
  public:
    Constrainable<Real> mass;
    Constrainable<Vector&> velocity;
    Constrainable<Vector&> force;
};

ParticleObject::ParticleObject ()
{
  assert( velocity == derivative(position)
          && force == mass * derivative(velocity) );
}
```

Two comments here: (a) recall that the `position` property is inherited from `AffineGeometricObject`, and (b) the `&&` operator takes two constraints and forms a single conjunction constraint with the obvious semantics.

For convenience, TBAG uses a slightly different formulation of `ParticleObject` that contains a set of component force constrainables and automatically maintains the constraint that the net force equals the sum of the component forces. These component forces may come from a variety of physical sources. For instance, there is a utility function `spring_force` that computes the force generated at one end of a spring given that end's position, the spring's stiffness, rest length, and the position of the spring's opposite end.

```
Vector&
spring_force(Point& pos, Real stiffness,
             Real rest_length, Point& opp_end)
{
  Vector& sep = opp_end - pos;
  Real length = vector_magnitude(sep);
  return stiffness * sep * (rest_length - length) /
         length;
}
```

Another simple component force generator is drag, which is based on a particle's velocity and a drag coefficient.

```
Vector&
drag_force(Real drag_coeff, Vector& vel)
{ return -drag_coeff * vel; }
```

Once the header files for these force-producing functions are processed by the overloading tool described in Section 2.2.2, the generated overloadings may be applied to constrainable arguments. In particular, for the `spring_force` function, the particle's position and the position of the spring's opposite end are usually constrainables.

Figure 7 shows a spring toy that was constructed out of a few simple geometrically realized particles and springs. Any of the balls may be grabbed, moved, and even thrown. They then bounce around in a physically realistic looking manner. The posts may also be moved, and are constrained so that their bases remain in the ground plane. The forces acting on each ball are gravity, drag, and two spring forces. Figure 8 shows a construction set that allows the user to add balls and anchors and place springs to connect balls to each other or to anchors.

In these examples, the user may grab and pull on a ball. Rather than overriding the Newtonian constraint, we chose to implement such manipulation of physical objects by means of a transient anchor-and-spring pair. When a physical object is grabbed, an anchor and spring are created and added to the physical object's geometry and the resulting spring force is added to the set of component forces.

In order to handle systems of ODEs, the basic constraint engine [19] was extended with a facility for detecting cyclic constraint subgraphs and then attempting to apply constraint-specific resolution techniques. Extraction of ODE systems is the only such resolution technique we have added. Systems of linear or non-linear equations could be handled in the same way. Briefly, ODE resolution works as follows: each cycle in the identified constraint subgraph is checked to make sure it has at least one derivative constraint for which the integration function has been chosen to satisfy it (i.e., the derivative constraint is being "executed backwards"). If not, cycle resolution fails. Otherwise, a system of simultaneous ODEs is constructed and the identified constraint subgraph is replaced by one that executes the ODE solver and extracts the results for each involved constrainable.

5.1 Multi-type Differentiation

TBAG's notion of derivative, and hence integration and ODE solution, are not limited to Real-valued constrainables. Rather, derivatives are defined on many types, including `Real`, `Point`, `Vector`, `Quaternion`, and `Transform`. These types are supported, and more may be added, via an extension mechanism in which the basic operations underlying differentiation and integration are defined. For each such type *T* (e.g., `Point`), a "delta type" *T'* (e.g., `Vector`) is identified, together with functions including subtraction (mapping two *T* values to a *T'* value), addition (mapping a *T* value and a *T'* value to a *T* value), and scalar multiplication (mapping a real number and a *T'* value to a *T'* value). The definition of these operations for the `Quaternion` is particularly interesting. Rather than using subtraction, addition and scaling on quadruples, we use quaternion division, multiplication, and exponentiation, respectively. We find these definitions to yield much more useful results. For instance, the virtual trackball algorithm used by TBAG's geometry viewer is expressed in terms of a quaternion derivative constraint and solved by the ODE engine via quaternion integration. Finally, the operations on affine transforms are based on corresponding operations on scale, shear, rotation and translation components.

The sets of functions needed to support differentiation also suffice to support multi-type interpolation. Thus, TBAG supports a very general form of linear interpolation: given two values of type *T*, say v_0 and v_1, and a real number *t*, compute a new value corresponding to $v_0 + (v_1 - v_0) * t$. The +, -, and * operations refer to the corresponding operations on types *T* and *T'*. Interpolation on the above mentioned types is thus supported. System extenders interested in adding operations such as 3D "morphing" would simply need to provide the appropriate +, -, and * functions on the Geometry type.

5.2 Incremental Evaluation

Sampling of constrainables that are driven by integration or ODE solution is side-effect-free, like all sampling in TBAG. A simplistic implementation would be to perform numerical iteration from the initial time and value for each sample, but of course such an approach would be too expensive. Instead, in our implementation, integral and ODE behaviors transparently cache the time and value of the latest sample. Because behaviors are almost always successively sampled at times that differ by a small amount, the iterative numerical algorithm typically requires only a few iterations per behavior sample. Of course, the use of state is far from unusual in uses of differential equation solvers. The difference in TBAG is that the use of state is automatic and transparent.

6 A Collection of TBAG Applications

A variety of applications have been written using the TBAG system. This section briefly describes some of them.

The **SoundScape** application in Figure 9 presents a three dimensional landscape of 3D icons, each of which represents a sound being emitted from that point in space. The user may manipulate two microphones (representing the user's left and right ears) and any of the 3D icons, resulting in the user perceiving the sounds in the vicinity of the left and right microphones. In a distributed setting, two separate users can each grab one microphone, thus effecting what they and the other user are hearing in one of their two ears.

The **EagleWatcher** application in Figure 10 (inspired by an interactive animation done at Brown University) presents four "watchers" whose eyes continually track the position of the eagle. The gaze is maintained as the eagle and watchers move. This application was constructed by establishing a constraint relationship between the watcher's head and eye angle, the position of the watcher and the position of the eagle. Additionally, each watcher is emitting a voice, and the user of the application hears the voices that the eagle might hear through its ears, based upon the distance of each watcher to the eagle.

The **ColorView** application shown in Figure 11 (also inspired by an example done at Brown University) presents three interactive views of a color based on its RGB representation: a color cube, three 3D sliders, and three 2D GUI sliders (not shown). Any view may be controlled with the mouse, causing the other two change accordingly. In the color cube case, the RGB components are constrained to the *relative* position of the ball within the cube. As a pleasantly surprising consequence of multi-directionality, the color will change continuously if the cube frame is spun and the ball grabbed and held still..

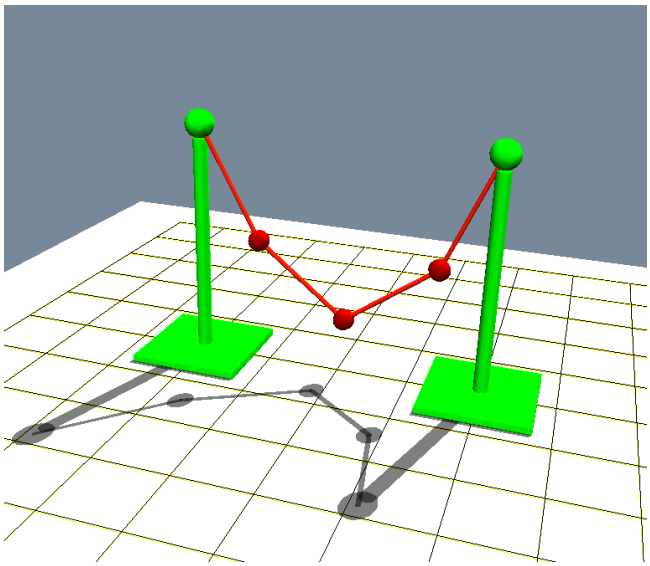

Figure 7. A Spring Toy

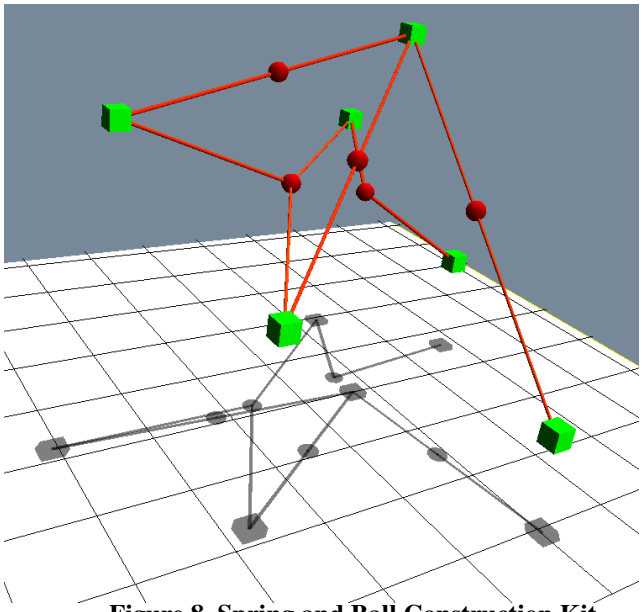

Figure 8. Spring and Ball Construction Kit

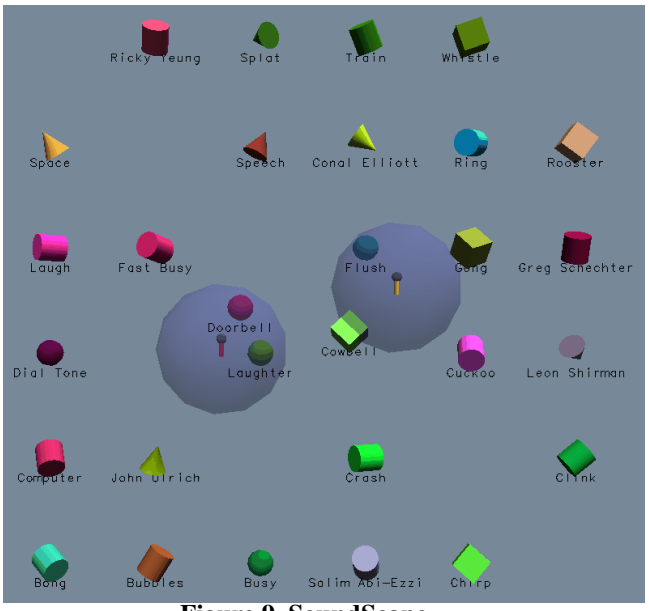

Figure 9. SoundScape

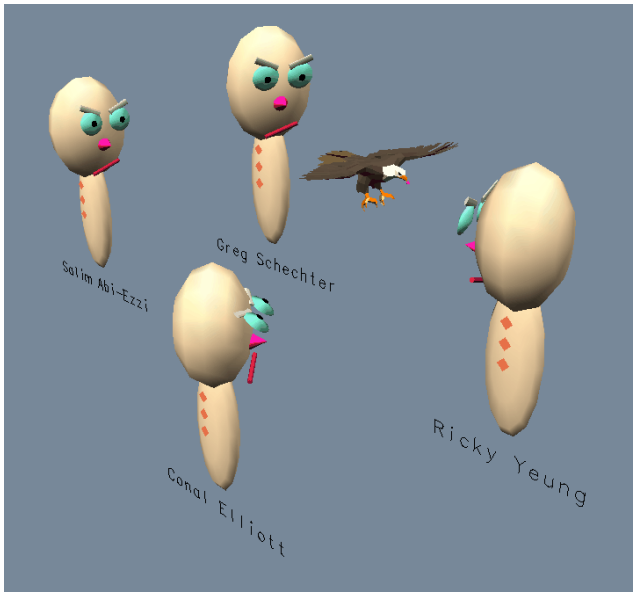

Figure 10. EagleWatcher

The **MortgageTool** application in Figure 12 uses multidirectional constraints to allow flexible analysis of the variables that go into a mortgage. The user may vary any subset of the Principal, Interest Rate, Number of Periods, and Payment per Period variables, resulting in changes to the remaining variables. (Contrast this with a traditional spreadsheet based approach, where the designer of the spreadsheet is forced, in designing the spreadsheet, to specify which variables are to be input variables, and which are to be output variables.) MortgageTool provides a direct manipulation interface in which the bars of the chart may be directly grabbed and moved. The third dimension is exploited to provide a side-by-side visual comparison of independent scenarios.

The application shown in Figure 13 provides an animated view of various sorting algorithms. Shown here is a MergeSort. Comparisons and swaps are animated and have sound effects.

Finally, Figure 14 shows an implementation of a Xerox PARC-style ConeTree [4]. This fully animated version (with slow-in/slow-out animation) was implemented in less than one day by a summer intern.

7 Related Work

3D Programming Systems. Also aimed at simplifying construction of interactive 3D programs, Inventor [22] provides a gallery of standardized interaction techniques. Following PHIGS PLUS, Doré, and others, Inventor adopts a procedural, state-based, and discrete approach. Unlike TBAG, these systems are heavily order-dependent and reliant on side-effects (similar to PHIGS structure editing), to achieve application goals.

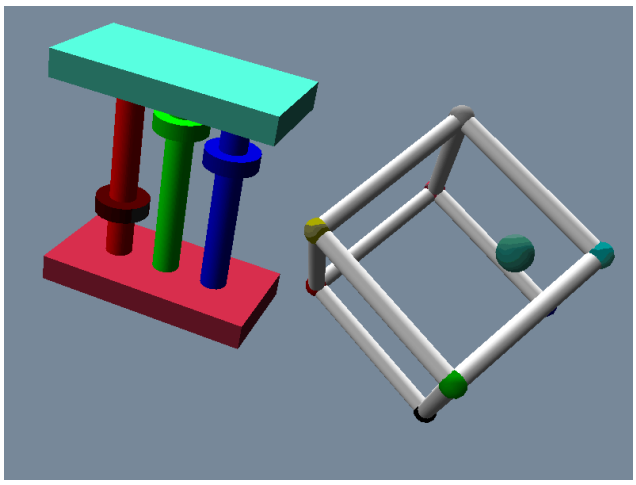

Figure 11. ColorView

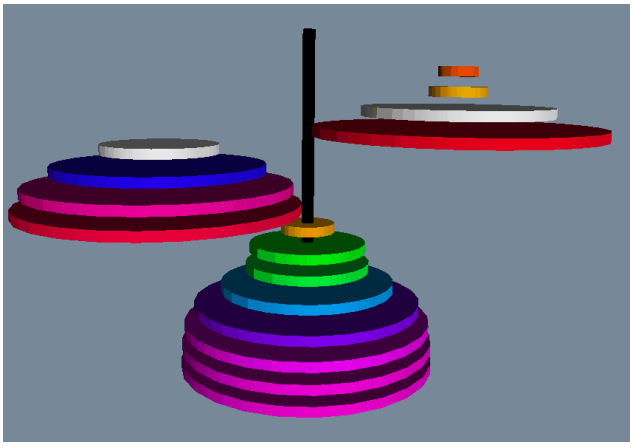

Figure 13. Sorting Algorithm Animation

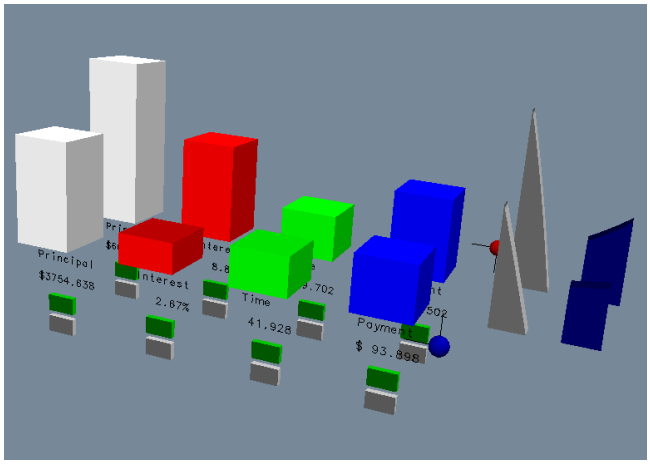

Figure 12. MortgageTool

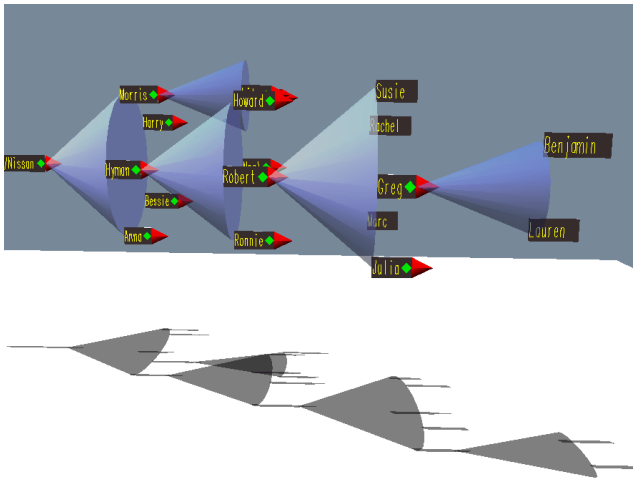

Figure 14. Xerox ConeTree

Mirage [24] is a high level 3D object-oriented graphics system that supports a hierarchical temporal coordinate system, but, unlike TBAG, does not treat time-varying values as a first-class notion.

The UGA [29] work appears to be the first 3D programming framework that supports direct expression of time-varying values as functions of time and input. While both UGA and TBAG are fundamentally continuous, TBAG has focused more on the integration of high-level data types while UGA has focused more on language mechanisms such as delegation hierarchies.

Animation Systems. Traditional animation systems support mostly key-frame animation. This discrete frame-based approach is also reflected in many existing animation languages. Descriptions of systems that embed animation capability within a general-purpose programming language can be found in Reynolds' [18] and Thalmann's [25] work. Some interesting variations include S-Dynamics [21], which allows time dependent parametric descriptions of actions, and Arya's lazy functional approach to animation, based on infinite sequences [1]. TBAG, on the other hand, supports a fundamentally continuous model of animation, and does so in the context of a production programming language (C++).

Applications of Constraints to Graphics. Constraint technology has long been applied in 2D drawing systems, such as Sketchpad [23] and Juno [16], and in 2D user interfaces, such as ThingLab II [14] and Garnet [15]. In 3D graphics systems, the use of data-flow or unidirectional dependency networks is common; examples can be found in UGA [29], CONDOR [13], AVS [26], and ConMan [11]. In contrast to these systems, TBAG combines the use of multi-directional constraints with the ability to involve high level types in continuous constraint relationships, and does so via a simple syntax.

Developed independently, VB2 [9] also applies the SkyBlue incremental constraint resolution algorithm to 3D graphics. In contrast to TBAG, however, VB2 does not support a continuous time model, high level data types, or a succinct syntax for building up constraint networks.

QOCA [12] independently used overloading to simplify expression of constraints. Unlike TBAG, QOCA supported only Real-valued constraints and supports overloading of the four basic arithmetic operators.

Much good work has gone into extending the might of numerical constraint solvers and applying them to interactive graphics (e.g. [28,2,7,8]). Our own work is complementary, exploring the application of constraints to a multitude of types (including very high level types), and providing an almost invisible programming interface to it.

The Animus system [5] embodies some of the earliest research done on considering time in constraint programming. Its notion of temporal constraints is based on a discrete history mechanism, as opposed to TBAG's fundamentally continuous approach. Consequently, ODE-based applications must express the numerical integration algorithm used (Euler's in the examples given) rather than simply the differential equations themselves.

8 Conclusions

This paper has presented TBAG, a paradigm and toolkit for rapid prototyping of interactive, animated 3D graphics programs. The fundamental aspects of TBAG, high level graphical abstract data types, and explicit functions of time, are applied broadly, treating, e.g., points, planes, colors, transforms, geometry, lights, shadows, and sound, in a consistent manner. The immutability of these types and functions allows for efficient automatic memory management and distributed execution. Automatically generated overloading of existing functions and operators give rise to succinctly expressed interactive animations.

Our current direction is in applying these same concepts uniformly to other media types, in pursuit of a coherent framework for distributed integrated media.

9 Acknowledgments

The authors would like to thank Leon Shirman, Srikanth Subramaniam, and Michael Deering for their contributions to the ideas and implementation of TBAG, Tom Meyer and Ajay Sreekanth for stress-testing TBAG and developing sample applications during their summer internships, and Matt Peréz for his support of this work.

Finally, we would like to thank the Brown University Graphics Group under Andries van Dam and John Hughes for their pioneering work on UGA, for a very fruitful collaborative relationship, and for comments on an earlier draft of this paper. In particular, work with Matthias Wloka contributed toward our current notion of behaviors; several discussions with John Hughes helped to correct and refine our current and future work with derivatives, and the whole group worked with the TBAG system and made extensive comments on its design.

10 References

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

[2] Ronen Barzel and Alan H. Barr. A Modeling System based on Dynamic Constraints. Proceedings of SIGGRAPH '88. In *Computer Graphics* 22, 4 (August, 1988).

[3] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), October, 1981.

[4] S.K. Card, G.G. Robertson, and J.D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 181-188. 1991.

[5] Robert A. Duisberg. Animated Graphical Interfaces using Temporal Constraints. In *CHI'86 Conference Proceedings*, pages 131-136, Boston, April 1986.

[6] *Graphics Library Programming Guide*, Silicon Graphics Computer Systems, Mountain View, Calif., 1991.

[7] Michael Gleicher and Andrew Witkin. Differential manipulation, *Graphics Interface*, June 1991.

[8] Michael Gleicher and Andrew Witkin. Through-the-lens Camera Control. Proceedings of SIGGRAPH '92. In *Computer Graphics*, 26, 2 (July, 1992).

[9] Enrico Gobbetti, Jean-Francis Balaguer, and Daniel Thalmann. VB2: An Architecture for Interaction in Synthetic Worlds. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 167-178, November, 1993.

[10] 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.

[11] Paul E. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. Proceedings of SIGGRAPH '88. In *Computer Graphics* 22, 4 (August, 1988).

[12] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An Object-Oriented Architecture for Constraint-Based Graphical Editing. *Eurographics Object-Oriented Graphics Workshop*, pages 1-22, 1992.

[13] Michael Kass. CONDOR: Constraint-Based Dataflow. Proceedings of SIGGRAPH '92. In *Computer Graphics*, 26, 2 (July, 1992), 321-330.

[14] John H. Maloney, Alan Borning, and Bjorn N. Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *OOPSLA '89 Proceedings*, October 1989.

[15] Brad A. Myers, Dario A Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, November, 1990.

[16] Greg Nelson. Juno, A Constraint-Based Graphics System. Proceedings of SIGGRAPH '88. In *Computer Graphics* 22, 4, (August, 1988), 235-243.

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

[18] Craig W. Reynolds. Computer Animation with Scripts and Actors. Proceedings of SIGGRAPH '82. In *Computer Graphics,* 289-296.

[19] Michael Sannela. The SkyBlue Constraint Solver. TR-92-07-02, Department of Computer Science, University of Washington.

[20] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D Graphics in C++ – with an Object-Oriented, Multiple Dispatching Implementation. To appear in the proceedings of the *1994 Eurographics Object-Oriented Graphics Workshop*.

[21] *S-Dynamics*, Symbolics, Inc., Cambridge, MA. 1985.

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

[23] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Spring Joint Computer Conference*, pages 329-345, 1963.

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

[25] Nadia Magnenat-Thalmann and Daniel Thalmann. *Computer Animation: Theory and Practice*. Springer-Verlag, Tokyo, 1985.

[26] Craig Upson, Thomas Faulhauber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, pages 30-42, July, 1989.

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

[28] Andrew Witkin, Kurt Fleischer, and Alan Barr. Energy Constraints on Parameterized Models. Proceedings of SIGGRAPH '87. In *Computer Graphics* 21, 4, (July, 1987).

[29] Robert C. Zeleznik, D. Brookshire Connor, Andries van Dam, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry E. Kaufman, and John F. Hughes. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. Proceedings of SIGGRAPH '91. In *Computer Graphics* 25, 4, (August, 1991), 105-112.