A Brief Introduction to ActiveVRML

Conal Elliott

February, 1996

Technical Report
MSR-TR-96-05

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Introduction

This paper introduces the *Active Virtual Reality Modeling Language* (ActiveVRML), a modeling language for specifying interactive animations.

To allow the creation of interactive animations to be as natural as possible, ActiveVRML is based on a simple and intuitively familiar view of the world; that is, as a hybrid of continuous variations and discrete events. For example, the behavior of a bouncing ball consists of continuous trajectories and discrete collisions. Trajectories cause collision events, and collision events cause new trajectories.

Using ActiveVRML, one can create simple or sophisticated animations without programming in the usual sense of the word. For example:

- Although many frames are generated in presenting an animation, the author is freed from any notion of sampling or frame generation, but rather describes how various animation parameters vary continuously with time, user input, and other parameters.
- An author describes events influencing an animation and the effects of these events on the animation. The author is freed from the programming mechanics of checking for events and causing the effects to happen.
- Although animations involve an extremely high degree of simultaneity (concurrency), the author is freed from such programming issues as multi-threading.
- Linguistically, there are no statements (commands) that are executed for their effect, but rather expressions that are analyzed for their value. ActiveVRML uses this approach to make specifying animations as natural as possible, while simultaneously retaining maximal opportunities for optimization.[1]

While ActiveVRML is modeling language, it exploits three of the key ideas that give programming languages their tremendous power:

- *Composition*. Animations are built of simpler animations in a modular, building block style. By applying composition repeatedly, complex animation can be constructed, while each layer of description remains manageable.
- *Parameterization*. Families of related animations can be defined in terms of parameters of any kind, including other animations.
- *Scoped naming*. Animations and animation families can be given names to facilitate readability and convenient reuse. The naming of an animation can be explicitly

[1] In programming language terms, ActiveVRML is a *declarative*, rather than *imperative*, language.

limited, or *scoped*, so as not to conflict with possibly unrelated uses of the same name elsewhere in a description.

ActiveVRML applies these principles pervasively to all types of static models, continuous behaviors, and discrete events.

To make the discussion of ActiveVRML more concrete, the first few sections of this paper use a running example—a solar system that begins as a single static planet, and then adds animation, other planets, and sound.

The remainder of this paper is organized as follows. We first outline the media types and operations. Next, we describe how ActiveVRML complements other Internet-standard file formats by supporting importation. We then illustrate the keys ideas of composition, parameterization, and scoped naming. Next, we introduce *behaviors*, which are time-varying values of all types. We then show how to add spatialized sound to a model. Next, we explain *reactivity* and the various kinds of events that support reactivity. We then describe support for user interaction. Next, we illustrate the principle of time transformation, which provides temporal modularity. We then briefly describe the built-in support for behaviors defined in terms of rates of change. Finally, we develop as an extended example, a collection of balls bouncing around in a box.

Overview of Supported Media Types

This section provides a brief overview of the media types supported in ActiveVRML. (All of the types and operations are time-varying.)

- *3-D geometry*. Supports importation, aggregation and transformation. Also supports texture mapping of interactive animated images, manipulation of color and opacity, and embedding of sounds and lights.
- *Images*. Provides infinite resolution and extent images. Supports importation, 2-D transformation, opacity manipulation, and overlaying. Also supports rendering an image from a 3-D model and rendering an image out of rich text. Even geometrical and image renderings have infinite resolution and extent, since discretization and cropping are left to the display step, which is always left implicit.
- *Sound*. Rudimentary support for importing, manipulating, and mixing sounds. Also, *sonic rendering* of 3-D models; that is, geometric models may be listened to as well as looked at. Conceptually infinite sampling rate and sample precision.
- *Montages*. Composite 2 ½-D images, supporting convenient, multi-layered cel animation.
- *2-D and 3-D points and vectors*. Operations include vector/vector and point/vector addition, point subtraction, scalar/vector multiplication, and dot and cross products. Also supports construction and deconstruction in rectangular and polar/spherical coordinates.
- *2-D and 3-D transforms*. Supports translate, scale, rotate, shear, identity, composition, inversion, and matrix-based construction. Can be extended to non-linear deformations, and so forth.

- *Colors*. Various constants, construction, and deconstruction in RGB and HSL color spaces.
- *Text*. Rudimentary support for formatted text, with color, font family, optional bold, and italic. If there are Internet standards for rich text, then we would like to support importation as well.
- *Miscellaneous*. Support for numbers, characters, and strings.

Embracing Existing Formats

There is an enormous amount of raw material available today, both commercially and freely on the Internet, that can be used as a starting point for constructing interactive animations. This material is in files of many different formats representing geometry, images, video, sound, animation, motion paths, and so forth. ActiveVRML works with these representations directly, rather than requiring authors to create raw material specifically for ActiveVRML, or even converting existing material into a new format.

For our solar system, we start with a VRML 1.0 model of a unit sphere and an earth texture in GIF format. We import this content into ActiveVRML by means of `import`, and name the results for later use.[2]

```
sphere = import("sphere.wrl");
earthMap = import("earth-map.gif");
```

Each of these two lines is a *definition*, which both introduces a new name and provides an expression for the value of that name. The modeling notion of definition differs from the programming notion of *assignment*, in that the association between name and value established by a definition holds throughout a model's lifetime. Authors, readers, and automatic optimizers can thus know from seeing a definition like the first one above that `sphere` will always be the suggested imported model.

All names are typed, but types are almost always inferred automatically by ActiveVRML, and so rarely need to be specified explicitly. These two definitions implicitly declare `sphere` to be of type **`geometry`**, and `earthMap` to be of type **`image`**.

Compositional Specification

As mentioned in the introduction to this paper, *composition* is the building-block style of using existing models to make new models, combining the resulting models to make more new models, and so on.

To start building our earth geometry, we apply the earth texture to our earth sphere. We begin by making a texture:

[2] Geometry and image importation produces additional information beyond the geometry and image values themselves. We are omitting these values for brevity.

```
earthTexture = texture(earthMap);
```

We then apply the texture to the unit sphere:

```
unitEarth = earthTexture(sphere);
```

In our solar system, we will take the Sun's radius to be one unit, and the earth's to be half as big. Given the texture-mapped unit sphere, we first make a transform that scales by one half, uniformly.

```
halfScale = scale3(0.5);
```

Now we can form the reduced sphere by applying the `halfScale` transform to the texture-mapped unit sphere:

```
earth = apply(halfScale, unitEarth);
```

Next we want to reposition the earth, so that it will apart from the sun. We make a translation transform and then apply it to the earth:

```
moveXby2 = translate(2,0,0);
movedEarth = moveXby2(earth);
```

Giving names to transforms, textures, and geometric models at every step of composition leads to descriptions that are tedious to read and write. In ActiveVRML, naming and composition are completely independent, so the author is free to choose how much and where to introduce names, based on the author's individual style and intended reuse.

For example, we can name only the imported sphere and texture and the complete moved earth, as in the following description, which is equivalent to the previous one but does not introduce as many names:

```
sphere = import("sphere.wrl");
earthMap = import("earth-map.gif");

movedEarth =
  apply(translate(2,0,0),
   apply(scale3(0.5),
    texture(earthMap)(
     sphere))));
```

Next we build a model of the sun. No transformation is required, but we do want it to be yellow:

```
sun = diffuseColor(yellow)(sphere);
```

To complete our first very simple solar system, we simply combine the sun and moved earth into one model, using the infix `union` operation, which takes two geometric models and results in a new, aggregate model.

```
solarSystem1 = sun union movedEarth;
```

Scoped Naming

Naming is useful for making descriptions understandable and reusable, but can easily cause clutter. When intermediate animations are named and then used in only one or a few animations (as might be the case of `sun` and `movedEarth` above), they can interfere with available choices for intermediate names in other animations. While this clutter is not a problem with very simple animations described and maintained by a single author, it can become a serious obstacle as complexity grows and separately authored animations are combined to work together.

The solution to name clutter is to explicitly limit the scope of a name's definition. In our example, we will leave the `sphere`, `earthMap`, and `solarSystem1` definitions unscoped, but limit the scope of the `sun` and `movedEarth` definitions.

To limit the scope of a collection of definitions to a given expression, use the form

let *definitions* **in** *expression*

(In addition to the given expression, the scope of the definitions include the bodies of all of the definitions themselves, to allow for mutual recursion.)

```
solarSystem1 =
let
    movedEarth =
      apply(translate(2,0,0),
       apply(scale3(0.5),
        texture(earthMap)(sphere))));

    sun = diffuseColor(yellow)(sphere);
in
    sun union movedEarth;
```

The scope of `movedEarth` and `sun` is the expression in the last line of this definition of `solarSystem`. Any other potential uses of the names `movedEarth` and `sun` would not refer to the scoped definitions above.

Parameterization

It is often desirable to create several animations that are similar but not identical. If such models differ only by transformation—for instance, if they are translations and orientations of a single model—the composition approach is helpful. In general, however, reuse with transform application (which corresponds to the *instancing* facility commonly found in graphics modeling and programming systems) is a very limited technique.

ActiveVRML goes far beyond instancing by providing a simple but extremely general and powerful form of *parameterization*. Families of related animations can be defined in terms of parameters of any kind, including other animations.

As an example of parameterization, suppose that we want a variety of simple solar systems differing only in the sun color and an angle of rotation of the earth around the sun. Each of these solar systems has its own color and own rotation angle, but in all other ways is identical to its other family members. We define such a family as follows. (Note that `sunColor` and `earthAngle` are parameter names that refer generically to the color and angle that distinguishes one simple solar system from another.)

```
solarSystem2(sunColor, earthAngle) =
let
    movedEarth =
      apply(rotate(yAxis, earthAngle),
       apply(translate(2,0,0),
        apply(scale3(0.5),
         texture(earthMap)(sphere)))));

    sun = diffuseColor(sunColor)(sphere);
in
    sun union movedEarth;
```

To instantiate a solar system from this family, apply `solarSystem2` to a color and an angle. For instance,

```
solarSystem2(yellow, 0)
```

Behaviors

Up to this point, our examples have described *static models*—that is, models that do not vary with time. These models were built compositionally, from static numbers, colors, images, transforms, and other models. In ActiveVRML, one can just as easily express *behaviors*, that is, time-varying values of all types, with static values being just degenerate versions of the general case.

The simplest non-static behavior is `time`, which is a number-valued behavior that starts out with value zero and increases at a rate of one unit per second.

As a simple example of a compositionally defined behavior, the following expression describes a number-valued behavior that starts out with value zero and increases at a rate of 2π per second:

```
rising = 2 * pi * time;
```

The use of `time` here refers to a *local*, not a *global* notion of time. Just as geometric models are generally specified in spatial local (or *modeling*) coordinates, behaviors of all types are generally specified in local temporal coordinates, and are then subjected to temporal transformation, as discussed in the section "Time Transforms," and combined with other, possibly temporally transformed, behaviors.

We can use this number behavior to describe a time-varying uniform scaling transform that starts as a zero scale and increases in size:

```
growing = scale3(rising);
```

And we can use this `growing` behavior to describe a geometry-valued behavior, that is, a 3-D animation, such as solar system growing from nothing:

```
growingSolarSystem1 = apply(growing, solarSystem1);
```

As always, intermediate definitions are optional; we could just as well use:

```
growingSolarSystem1 =
  apply(scale3(2 * pi * time), solarSystem1);
```

With a slight variation, we could have the scale go back and forth between 0 and 2:

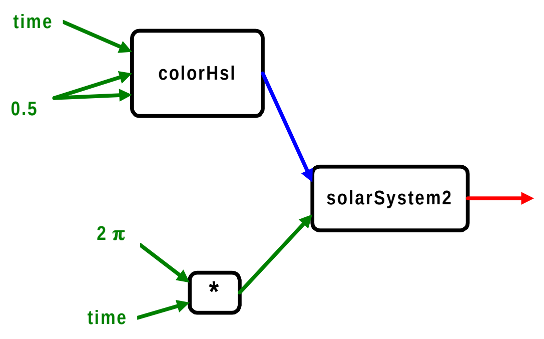
```
pulsating =
  apply(scale3(1 + sin(time)), solarSystem1);
```

We can also apply our `solarSystem2` family, defined above, to behavior arguments to create time-varying solar systems, as in the following example in which the sun color runs through a variety of hues while the earth rotates around the sun.

```
animatedSolarSystem2 =
  solarSystem2(colorHsl(time, 0.5, 0.5), 2 * pi * time)
```

Behaviors as Data Flow

For some people, it is helpful to visualize behaviors as data flow graphs. For example, the `animatedSolarSystem2` behavior above can be illustrated as in the figure below. Note that, unlike traditional data flow, behaviors describe a *continuous* flow of values, not a discrete sequence.



Data flow diagrams, while somewhat helpful for illustrating simple non-reactive behaviors, are much weaker than what can be expressed in ActiveVRML, because of both reactivity and time transformability.

More Parameterization

We would now like to enrich our solar system in two ways: by making the earth revolve around its own axis, as well as rotate about the sun, and by adding a moon that revolves about its axis and rotates around the earth. Parameterization allows us to capture the similarities between moon and earth, while allowing for their differences.

We start with a simple definition that rotates a given model with a given period:

```
rotateWithPeriod(geo, orbitPeriod) =
    apply(rotate(yAxis, 2 * pi * time / orbitPeriod), geo);
```

We use `rotateWithPeriod` to create a revolving earth and moon and as a building block for the following definition, which puts models into orbit:

```
orbit(geo, orbitPeriod, orbitRadius) =
  rotateWithPeriod(apply(translate(orbitRadius, 0, 0), geo),
                   orbitPeriod)
```

We can now define our extended solar system:

```
solarSystem3 =
let
    // constants

    sunRadius = 1                            // size of the sun
    day = 3                                   // seconds per day

    earthRadius         = 0.5 * sunRadius    // size of earth
    earthRotationPeriod = 1 * day
    earthOrbitRadius    = 2.0 * sunRadius
    earthOrbitPeriod    = 365 * day

    moonRadius          = 0.25 * earthRadius // size of moon
    moonRotationPeriod  = 28 * day
    moonOrbitRadius     = 1.5  * earthRadius
    moonOrbitPeriod     = moonRotationPeriod

    // sun is a yellow sphere
    // earth is a sphere with the earth-map texture
    // moon is a gray sphere

    sun   = apply(scale3(sunRadius),
              diffuseColor(yellow)(sphere));
    earth = apply(scale3(earthRadius),
              texture(earthMap)(sphere);
    moon  = apply(scale3(moonRadius),
              diffuseColor(rbgColor(0.5,0.5,0.5))(sphere))

    // define the relationships between and the motions of the bodies

    moonSystem = rotateWithPeriod(moon, moonRotationPeriod)
    earthSystem =
      RotateWithPeriod(earth, earthRotationPeriod) union
       orbit(moonSystem, moonOrbitPeriod, moonOrbitRadius)
    sunSystem =
          sun union
           orbit(earthSystem, earthPeriod, earthOrbitRadius)
in
        sunSystem
```

Adding Sound

We will now add sound to our solar system example by having the earth emit a "whooshing" sound[3]. The sound will come from the earth, so as a user moves around in the solar system or as the earth moves around, the user will be able to maintain a sense of the spatial relationship, even when the earth is out of sight. Moreover, if the moon is making a sound as well, the user will hear both sounds appropriately altered and mixed.

All that is necessary to add sound is to change the earth to include a spatially embedded sound; we modify `earth` in the `solarSystem2` definition as follows:

```
earth =
    apply(scale3(earthRadius), texture(earthMap)(sphere))
  union
    soundSource3(import("whoosh.au" ));
```

The `soundSource3` function used here places a sound at the origin in 3-D space, converting it into a geometric model, which can then be transformed and combined with other geometric models.

We can also make sound attributes vary with time. For example, we can adjust the earth sound's pitch so that it fluctuates during the day, as in the following definition. The formula used with `pitch` below causes the pitch factor to vary between 0.5 and 1.5 and back through the course of a day.

```
earth =
    apply(scale3(earthRadius), (earthMap)(sphere)
  union
    soundSource3(
     pitch(sin(2 * pi * time /day)/2 + 1)(
      import("whoosh.au" ));
```

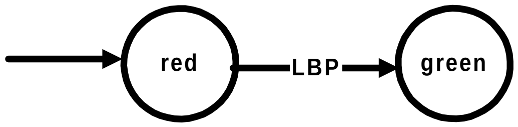
Reactivity

In the real world, as well as in computer games, simulations, and other applications of interactive animation, behaviors are influenced by *events*, and can be modeled as a series of events and reactions (or *stimuli* and *responses*). In this paper, we refer to behaviors that react to an event as *reactive behaviors*.

[3] (In the time-honored science fiction-movie tradition of sounds in space.)

Simple Reactivity

As a very simple example of a reactive behavior, suppose that we want our solar system's base color to be red at first, but then become green when a user presses the left button on the mouse. We can illustrate this two phase reactive color as follows, where, for succinctness, `LBP` refers to the event of pressing the left button:



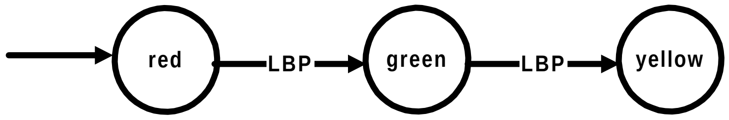
In ActiveVRML, this behavior is expressed as

```
twoPhase = red until LBP => green
```

In this example and the following ones, the behavior phases are static values. In general, however, they may be arbitrarily complex behaviors.

Chaining

When the user presses the left button, `twoPhase` turns from red to green, and stays green permanently; that is, it is no longer reactive. We can also specify a behavior that is still reactive in its second phase. For example, we can have the solar system's color change to yellow when the user presses the left button for the second time:

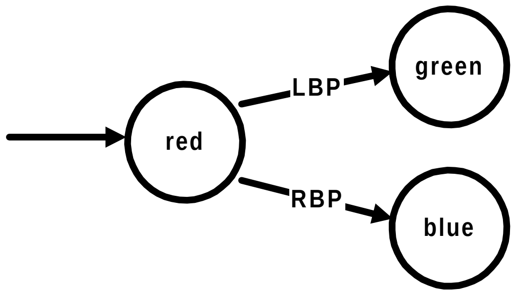


In ActiveVRML, this process is expressed as follows:

```
threePhase =
  red until
    LBP => (green until LBP => yellow)
```

Competing Events

In the `twoPhase` and `threePhase` examples, each phase was interested in at most one event (`LBP` or nothing). Often, a phase reacts to a number of different events, each leading to a different new phase. For instance, we can define a variation of `twoPhase` that also starts in the red phase, but will react to either a left or right button press with a different new behavior:



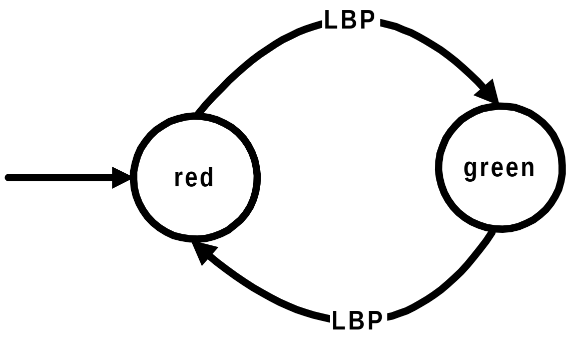
where RBP refers to our user's right button press event.

In ActiveVRML, this process is expressed as follows:

```
choose =
  red until
    LBP => green
  | RBP => blue
```

Repetition

Now suppose we want a color that switches back and forth between red and green at each button press, no matter how many times a button is pressed. Describing this repetitive behavior by a chain of single-color phases, as with `twoPhase` and `threePhase`, requires a infinite chain. Fortunately, this infinite chain has a succinct description.



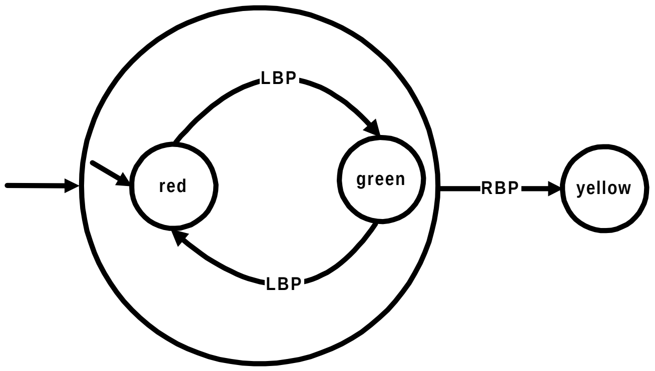
In ActiveVRML, this repetitive behavior is expressed as follows:

```
cyclic =
  red until
    LBP => green until
      LBP => cyclic
```

Note As illustrated in this example, ActiveVRML definitions may be self-referential.

Hierarchical Reactivity

In the previous three reactive behavior examples, each phase was a simple static color. In general, each phase of a reactive behavior can be an arbitrary behavior, even a reactive one. For example, we may want to present our user with the red/green cyclic behavior above only until the user presses the mouse's right button, at which time the color becomes permanently yellow.

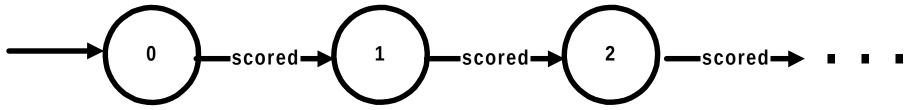


In ActiveVRML, this process is expressed as follows:

```
cyclic until
  RBP => yellow
```

Parametric Reactivity

Sometimes a reactive behavior goes through a sequence of phases that are similar, but not identical. For instance, a game may need to keep track of a player's score. Supposed we have already defined `scored` to refer to the event of a player scoring a point. (The subject of how events such as `scored` are defined is addressed later.) A score-keeping behavior can be illustrated as follows:



Each phase in this score-keeping behavior is similar in that its value is a static number. It is waiting for an occurrence of the `scored` event, at which time it will switch to a similar phase with one greater value. To define all of these phase behaviors at once, we describe the family parameterized by the only difference among them—the current score:

```
score(current) =
  current until
    scored => score(current+1)
```

The behavior that starts counting from 0 is expressed as follows:

```
scoreFromZero = score(0)
```

As always, we can limit the scope of the intermediate definition, even for parameterized definitions:

```
scoreFromZero =
let
    score(current) =
      current until
        scored => score(current+1)
in
    score(0)
```

Event Data

Some events have data associated with their occurrences. For example, each occurrence of a key press event has an associated character value. (It would be unwieldy to have a separate event associated with every key on a keyboard.)

As another example of events with data, we can generalize our score-keeping behavior so that each occurrence of the `scored` event could have its own number of points to be added to the total score. In the new version shown below, the event data generated by the `scored` event (number of points) is consumed by a parameterized behavior (`addPoints` below), which adds the number of points to the current score and continues counting.

```
score(current) =
let
    addPoints(points) =
      score(current+points)
in
    current until
      scored => addPoints
```

As mentioned in the previous section “Compositional Specification,” naming is optional. Even parameterized definitions can be replaced by the parameterized behavior itself, using the construct **`function (`***parameters***`).`** *Expression.* The following definition of `score` is equivalent to the previous one.

```
score(current) =
  current until
    scoreds => function (points). score(current+points)
```

The Varieties of Events

The preceding section illustrated a variety of ways to use events to describe behaviors in terms of other behaviors—that is, these behaviors are described *compositionally*. The next few sections examine how to describe the events themselves. As you may have guessed, in ActiveVRML, even events can be described compositionally.

External Events

Some events originate outside of ActiveVRML; for example, they can originate with a user, such as the left or right mouse button press events in some of our previous reactive examples.

Another example of an external event is a key press. Like a button event, a key press event can occur repetitively, but unlike a button event, key presses have associated data that indicates which character was pressed.

Predicate-based Events

Another kind of event is one in which a predicate (condition) about model parameters becomes true. For example, in the definition of `scoreFromZero` given above, the counting behavior goes on forever. We may, however, want to stop counting upon reaching some given maximum; that is, we may want to stop counting when the predicate `current = maxScore` becomes true for a given `maxScore`. Predicate-based events are written as **`predicate(`***condition_expression***`)`** as in the following replacement for `scoreFromZero.`

```
scoreUpTo(maxScore) =
let
    score(current) =
      current until
        scored => score(current+1)
      | predicate(current = maxScore)  => current
in
    score(0)
```

Note: In the context of a predicate, the equal sign (=) means equality, not definition.

Alternatively, we could define `scoreUpTo` in terms of the `scoreFromZero`.

```
scoreUpTo(maxScore) =
  scoreFromZero until
    predicate(scoreFromZero = maxScore)  =>  maxScore
```

These event conditions may be arbitrarily complex. As a slightly more sophisticated example, suppose we want a ball to respond to the event of hitting the floor. We'll define `center` as the (time-varying) height of the ball's center point, and `radius` as the ball's radius. We will consider the ball to be hitting the floor when two conditions are true: the bottom of the ball (that is, the center height minus the radius) is not above the floor, and the ball is moving in a downward direction (that is, the rate is less than zero).

In ActiveVRML, this event is expressed as follows:

```
hitFloor =
  predicate((center - radius <= floor) and (derivative(center) < 0))
```

Derivatives of this event are discussed later in this document.

Note The parentheses in this example are not required and are included for clarity only, since the syntactic precedence of **and** is weaker than that of inequality operators.

Alternative Events

Given any two events, we can describe the event that occurs when either happens. For example, the following describes either a left mouse button being pressed or our ball hitting the floor:

```
LBP | hitFloor
```

By repeatedly using the choice operator **|**, we can include as many component events as desired in the choice. For example:

```
LBP | hitFloor | predicate(scoreFromZero = maxScore)
```

Events with Handlers

Another way to build events is to introduce or enhance event data. For example, we may want an event that occurs whenever our user presses the left or right mouse button, and has value 1 if the left button is pressed and value 2 if the right button is pressed. First, we describe an event that occurs if the left button is pressed and has value 1:

```
LBP => 1
```

Then we describe a similar event based on the right button and having value 2:

```
RBP => 2
```

We then combine these two number-valued events into a single event:

```
buttonScore =  LBP => 1  |  RBP => 2
```

If an event already produces data, we can supply a way to transform the data into some other, more usable value. For example, we may want an event similar to `buttonScore`, but with values multiplied by 10. Rather than changing the definition of `buttonScore`, which may be needed elsewhere or may be out of our control, we make a new event by adding a multiply-by-ten event handler:

```
multiplyByTen(x) = 10 * x

buttonScore10 =
    buttonScore => multiplyByTen
```

We can do the same thing without introducing the `multiplyByTen` definition:

```
buttonScore10 =
  buttonScore => function (x). 10 * x
```

As another, simpler example of transforming event data, we may want to take a key press event and change all lowercase letters to uppercase.

```
keyPress => capitalize
```

Note It is no coincidence that the notation for alternative events (*`e|e'`*) and events with handlers (*`e=>f`*) is the same as introduced for reactive behaviors in the previous sections "Simple Reactivity" and "Event Data." The infix `until` operation used to express reactive behaviors applies to a behavior *b* and an event *e*, and yields a behavior that mimics *b* until the event *e* occurs, yielding a new behavior *b'*, at which time the until behavior starts mimicking *b'*.

User Interaction

ActiveVRML animations are intrinsically interactive, meaning that they know how to respond to user interaction events. We have already seen examples of events based on mouse buttons. Another form of input is a key press, which is similar to a button press but includes the generated character as event data.

Geometric user interaction is supported through an event where an animation is being probed. From the animation's viewpoint, the user's probe is a point-valued behavior that ActiveVRML breaks into a static point at the onset of probing and an offset vector behavior to show relative movement. These points and vectors are 2-D for probed images and 3-D for probed geometry.

Because there may be any number of transformed versions of an ActiveVRML animation coexisting at any time, there is no unique relationship between an animation and any given coordinate system, such as user coordinates. Thus, animations can only make sense of user input given to them within their own local coordinates. ActiveVRML automatically converts from the user's coordinates to the animation's own local coordinates.

For example, the following describes an image moving under user interaction:[4]

```
movingImage(startImage) =
  // Stay with startImage until clicked on.
```

[4] The event **andEvent (***e*,*e'***)** occurs when *e* and *e'* occur simultaneously. Its event data results from pairing the data produced from these two occurrences. Event handlers will then often destructure the resulting pair into its components and subcomponents, as in this example, in which the button press occurrence always generates the trivial data—which is written **()**—and the probe occurrence generates a point and vector behavior.

```
startImage until
  andEvent(leftButtonPress, probed(startImage)) =>
    function ((), (pickPoint, offset)).
     // Then make a version that moves with the offset
     // (given in modeling coords)
     let
         moving = apply(translate(offset), startImage)
     in
         // Then stay with the moving image until released.
         moving until
           // Then snap-shot the moving image and use to restart.
           snapshot(moving, leftButtonRelease) => movingImage
```

Time Transforms

Just as 2-D and 3-D transforms support spatial modularity in geometry and image behaviors, *time transforms* support temporal modularity for behaviors of all types.

For example, suppose we have a rocking sailboat expressed as follows:

```
sailBoat1 = apply(rotate(zAxis, sin(time) * pi/6),
             import("sailboat.wrl"))
```

If we want a slower sailboat, we could replace `sin(time)` with `sin(time/4)`, However, for reusability, we want instead to describe a new sailboat in terms of `sailBoat1`.

```
sailBoat2 = timeTransform(sailBoat1, time/4)
```

With this technique, we could define any number of coexisting similar sailboats, each having its own rate of rocking.

Differentiation and Integration

Because ActiveVRML time is continuous, rather than proceeding in a series of small jumps, it makes sense to talk about the rate of change of behavior of types such as number, point, vector, and orientation. For example, suppose that `moonCenter` is the time-varying position of the center of the moon. The moon's 3-D velocity vector (which is also time-varying) is expressed as follows:

```
derivative(moonCenter)
```

and the moon's 3-D acceleration vector is expressed as:

```
derivative(derivative(moonCenter))
```

Conversely, it is common to know the rate of motion of an object and want to determine the position over time. Given a velocity and an initial position, we could express the position over time as:

```
initialPos + integral(velocity)
```

It is often useful to specify the rate of motion of an object in terms of its own position. Suppose we have a goal, which may be moving, and we want to describe a point that starts at some initial position and always moves toward the goal, slowing down as it gets closer to the goal. The following definition describes this behavior:

```
pos = initialPos + integral(goal - pos)
```

This definition is equivalent to saying that the value of `pos` at the behavior's start time is `initialPos`, and that its velocity is `goal - pos`, which is in the direction of `goal`, relative to `pos`, with a speed equal to the square of the distance between `goal` and `pos`. If, for example, `pos` and `goal` coincide, then `pos` will not be moving at all.

Many realistic-looking physical effects can be described in this fashion, especially when the definitions are extended to use force, mass, and acceleration.

Note Integrals in this self-referential form are ordinary differential equations. Any number of such definitions may be expressed in a mutually recursive fashion to express systems of ordinary differential equations.

Implementations should take care to decouple the step sizes used in numerical integrators from that used for frame generation. There are a variety of numerically robust and efficient techniques, some of which adapt their step sizes to the local properties of the behavior being integrated.

Conclusion

In this paper, we have briefly introduced ActiveVRML, a language for modeling interactive, multimedia animations, and have illustrated some of ActiveVRML's expressiveness through a series of simple examples. We refer the interested reader to the *ActiveVRML Reference Manual* for more details.

Appendix A. An Extended Example

In this appendix, we present a larger *ActiveVRML* example, namely a collection of balls bouncing around in a box.

Geometry Importation

The first step in our example is to import the basic geometric components—a ball and a box. Each geometry importation yields both a (static) geometry and two 3-D points, representing a minimum bounding box for the imported geometry.

```
ball, ballMin, ballMax = import("ball.wrl");

rawBox, boxMin, boxMax = import("box.wrl");
```

We will use the ball geometry as is, but we need to make the box mostly transparent, so the bouncing balls inside will be visible.

```
box = opacity3(0.2)(rawBox);
```

One-Dimensional Bouncing

It will be useful to define a one-dimensional (number-valued) bouncing behavior, parameterized by lower and upper bounds, acceleration, and initial position and velocity. This bouncing behavior will be made up of an infinite sequence of phases, punctuated by bounce events. Each phase is parameterized by a initial position and velocity for that position, which start out as the overall initial position and velocity. The first bounce during a phase ends the phase, at which time the position and velocity are captured to provide the parameters of the next phase.

```
bounce1(min, max, accel, pos0, vel0) =
let
    // Describe one phase of behavior and transition to next, given
    // starting position and velocity.
    bouncePhase(newPos0, newVel0) =
    let
        // Start velocity at newVel0, and accelerate
        vel = newVel0 + integral(accel);
        // Start position at newVel0, and grow with velocity.
        pos = newPos0 + integral(vel);
        // Bounce event. Hits min descending or max ascending.
        bounce = predicate( (pos <= min and vel < 0)
                            or (pos >= max and vel > 0) )
    in
        // Follow this position phase until a bounce. Then snapshot
        // the position and the reversed, reduced velocity to get the
        // next starting position and velocity, and repeat.
    pos until
      snapshot((pos, -.9 * vel), bounce) => bouncePhase
in
    bouncePhase(pos0, vel0);
```

Three-Dimensional Bouncing

Next we will construct a 3-D bouncing behavior by appealing to the one-dimensional bouncing behavior for each of the three dimensions.

The minimum and maximum ball translations are determined from the box’s and ball’s minimum and maximum points, which were generated during importation. The ball’s minimum allowed translation is the one that when added to the ball’s minimum point puts it into contact with the box’s minimum point, and similarly for the maxima. These two observations lead to the following definitions for the minimum and maximum translation vectors:

```
ballTranslateMin = boxMin - ballMin;
ballTranslateMax = boxMax - ballMax;
```

Now we can define a bouncing ball geometry behavior, which is parameterized by the initial position and velocity.

```
bouncyBall(pos0, vel0) =
let
    // Appeal to the 1D version three times, ...
    xmin,ymin,zmin = XyzComponents(ballTranslateMin);
    xmax,ymax,zmax = XyzComponents(ballTranslateMax);

    x0, y0, z0  = XyzComponents(pos0);
    dx0,dy0,dz0 = XyzComponents(vel0);

    x = bounce1(xmin,xmax, 0  ,x0,dx0);
    y = bounce1(ymin,ymax, 0  ,y0,dy0);
    z = bounce1(zmin,zmax,-9.8,z0,dz0);
in
    // Use the results to translate the ball.
    apply(translate(x, y, z), ball)
```

It is a simple matter then to add a box, to get a single-ball version of our example:

```
bouncyModel1(pos0, vel0) =
  box union bouncyBall(pos0, vel0)
```

Many Bouncing Balls

Instead of just a single bouncing ball, we want an animation in which a user can cause any number of balls to be generated, all bouncing independently. To make this happen, we will define a second model, parameterized not by a single `(pos0,vel0)` pair, but rather by an event that produces `(pos0,vel0)` pairs, and adds a ball on each occurrence of the given event. This second model is the union of the box with a geometry composed of first no ball (the empty geometry), then one at the first occurrence of the given ball generator, then two at the second occurrence, and so forth.

```
bouncyModel2(ballGen) =
let
    balls = emptyGeometry until
              ballGen => function (pos0, vel0).
                          bouncyBall(pos0, vel0) union balls
in
    box union balls
```

Here is a brief explanation of how this definition works: At first, `balls` is the empty geometry. When `ballGen` occurs, its `(pos0,vel0)` pair is used to generate a single new bouncing ball, together with another instance of `balls`, which, as before, is empty until the first occurrence of `ballGen` (after this new `ball's` start), at which time this second instance of `balls` becomes a new bouncing ball together with a third instance of `balls`, and so on.

As a stylistic variation, we might factor our event processing into multiple phases: generation of `(pos0,vel0)`, by `ballGen`, conversion of `(pos0,vel0)` into a bouncing ball, by `bouncyBall`, and adding the rest of the balls, by a new function, `addRest`.

```
bouncyModel2(ballGen) =
let
    addRest(geom) = geom union balls

    balls =
      emptyGeometry until
        ballGen => bouncyBall => addRest
```

```
in
    box union balls
```

Note that the cascading effect of event data handlers. The **=>** operation associates to the left, so the handler line above is equivalent to

```
(ballGen => bouncyBall) => addRest
```

How might we define a ball generating event, as needed by `bouncyModel2`? There are many possibilities, but one very simple one is to wait for a button press event and then use the time of the button press to generate a pseudo-random position and velocity,

Vanishing Balls

With `bouncyModel2`, each new ball stays around forever once it comes into being. In this next variation, we will make each ball vanish (become the empty geometry) when it is picked. All we need to do is add another intermediate phase of event handling, `untilPicked`, that converts the newly created, permanent ball into a temporary one just before adding to the rest of the balls.

```
bouncyModel3(ballGen) =
let
    untilPicked(geom) =
      geom until
        andEvent(leftButtonPress, probe(geom)) => emptyGeometry

    addRest(geom) = geom union balls

    balls =
      emptyGeometry until
        ballGen => bouncyBall => untilPicked => addRest
in
```

```
box union balls
```