Release Terms

Preliminary Version

February 2, 1996

Microsoft Corporation

This draft specification is published by Microsoft for the purpose of helping to create an open standard. Independent implementations of this specification are expressly permitted by Microsoft irrespective of whether the specification contains, constitutes or reflects Microsoft patents or copyrights.

Microsoft anticipates that it will release a reference implementation in object and source code form. Microsoft expects to license this code in the following manner: (i) the object code may be reproduced and used without restriction or charge; and (ii) the source code may be reproduced and used for educational, non-commercial and internal use without charge, and for commercial use at a commercially reasonable charge.

CHAPTER

ActiveVRML Reference Manual

Introduction

This document presents a technical summary of the Active Virtual Reality Modeling Language (ActiveVRML). It provides a brief but reasonably precise definition of ActiveVRML version 1.0. Those seeking an introduction to ActiveVRML should consult the companion paper, *An Introduction To ActiveVRML*.

ActiveVRML is intended to provide a framework for constructing models that manipulate media including sound, two dimensional (2D) images, and three dimensional (3D) geometry. There are two characteristics that make ActiveVRML unique and especially well suited for this task: all values in ActiveVRML potentially vary with time, and values may change in reaction to events.

Every value in ActiveVRML may change with time. For example, there is an image type in ActiveVRML. An image object is not like a static photograph, but more like a video, continuously changing with time. Similarly, geometry in ActiveVRML is not like a static geometry model, but is (potentially) animated, moving, and reacting to events. This is an important principle in ActiveVRML; *every value* may change with time. Even simple objects, like numbers, may change with time. Values that can vary with time are called *behaviors* in ActiveVRML. A reactive behavior is one that (potentially) varies in response to events.

One way that values change with time is in response to particular events. For example, a user input event, or a mouse event. Events may be caused by internal events (to the ActiveVRML model) as well. For example, a particular number value being zero may cause an event.

Finally, ActiveVRML is a language for describing media via reactive behaviors. The language part of ActiveVRML is actually very small. The bulk of this document is spent describing a large collection of useful library functions for manipulating media types.

Expressions and Declarations

ActiveVRML is fundamentally very simple; it has just a handful of expression forms and only two forms for declaring identifiers. This section describes these forms. The power of ActiveVRML comes from the underlying model which includes time varying values, reactivity, and a rich set of media types. These are described in subsequent sections.

Literal and Constructor Expressions

Associated with most types in ActiveVRML is a literal or constructor form for each. Examples include 17 which represents a literal number, and [1, 2, 3], which uses the constructor form for lists to build a list of numbers. Allowable constructor forms are defined below in the sections defining each type.

Variable Expressions

An identifier in ActiveVRML is an arbitrarily long string of alpha-numeric characters beginning with a letter. Identifiers are case sensitive and there are some keywords (listed in Appendix A) that may not be used as identifiers.

Variables in ActiveVRML are statically scoped.

Application Expressions

An expression of the form *expression1 expression2* is an application expression and represents the value of applying the function value of *expression1* to the value of *expression2*. ActiveVRML is strict; that is, it obeys call by value semantics for argument evaluation. The order of the evaluation of arguments is not specified. Argument application associates left; for example, f(x)(y) equates to (f(x))(y).

Parenthetical Expressions

Parentheses may be used to group expressions and override operator precedence. Parentheses are also useful for improving the readability and presentation of ActiveVRML models. Operator precedence in ActiveVRML is listed in Appendix A.

If Expressions

A construct of the form

```
if expression1 then expression2 else expression3
```

is an **IF** expression. It represents the value returned by evaluating the boolean test of *expression1* and selecting *expression2* or *expression3* depending upon the true value of *expression1*. The types of the two branches of the **IF** expression are required to match (or unify). There is no corresponding **IF THEN** statement; all **IF** statements have both branches. Since ActiveVRML is functional (operations do not have side effects), such one-armed **IF** statements would not be very useful.

Let Expressions

A construct of the form

```
let
    declaration1;
        .
        .
        .
    declaration[n];
in
    expression
```

is a **LET** expression. It represents the value of an expression when evaluated in a context that includes the bindings for *declaration1* through *declarationn*. The **LET** expression is used to introduce a local scope. The declarations are evaluated simultaneously. The names are in scope of the right hand sides of the declarations, allowing for forward declarations and mutual recursion automatically. All of the declared identifiers are required to be distinct. The scope of the declarations is limited to the **LET** expression itself. The semicolon following the last declaration is optional.

Declarations

The simplest declarations declare an identifier to have a value:

```
identifier = expression
```

Or, they declare a function:

```
identifier(identifier, ..., identifier) = expression
```

The following is an example function declaration:

```
swizzle(seed) =
    if seed = 1 then
        1
    else if odd(seed) then
        swizzle(seed * 3 + 1) + seed
    else
        swizzle(seed / 2) + seed
```

This declares a function, *swizzle*, that takes one formal argument, *seed*. The function is recursive. All function declarations are assumed to be recursive. When you use the name of the function you are declaring within the expression, you are referring to the function itself, not a new or different function.

The following is an example variable declaration:

```
swizit = swizzle(27)
```

This declares the variable *swizit* to be the evaluation of the expression *swizzle*(*27*). We can illustrate scoping in **LET** expressions by combining these declarations along with a declaration for the predicate *odd* used in the declaration of *swizzle*:

```
let
    swizzle(seed) =
        if seed = 1 then
            1
        else if odd(seed) then
            swizzle(seed * 3 + 1) + seed
        else
            swizzle(seed / 2) + seed;
    odd(i) = (mod(i, 2) = 1);
    swizit = swizzle(27)
in
    swizit
```

Notice that the declaration for *odd* comes after its use in the declaration of *swizzle*. Since all of the declarations within a **LET** expression are assumed to be mutually recursive, this is legal. However, for better readability and because the declarations are not truly mutually recursive, the definition of *odd* should probably appear first.

Within the scope of the **LET** expression, three identifiers are declared, *swizzle*, *odd*, and *swizit*. Beyond the scope of this expression, the three declarations are not available. The value of the **LET** expression is the value of *swizit* which evaluates to 101440.

In addition to these simple forms of variable and function declarations, it is also possible to use pattern matching to specify the destructuring of values within a declaration. This is described later in this document.

In addition to the local declarations in an ActiveVRML file, it's possible to augment the environment with declarations from a second file with the **use** keyword:

```
use (pathname)
```

This keyword must be used as a top level declaration.

Basic Types

ActiveVRML includes a very powerful and useful typing system. Each expression and declaration in ActiveVRML is given a type either explicitly by the user, or implicitly by the system. Consider the following example declaration:

```
successor(nat) = nat + 1
```

ActiveVRML assigns *successor* the type *number -> number*, meaning it will map any value of type number to another value of type number. This typing is *strong* in the sense that ActiveVRML will catch all type errors during authoring. It is also convenient; the user did not have to explicitly give the type as the system inferred it.

Finally, types are *polymorphic*, meaning that a given type may stand for many different type instances. Consider the following declaration:

```
nada(val) = val
```

When applied, *nada* will return its actual argument unchanged. Thus *nada*(*3*) evaluates to the number 3 and *nada*(*"hello"*) evaluates to the string "hello". The type that ActiveVRML infers for nada is polymorphic: α -> α . Here α is a *type identifier* and may be replaced everywhere uniformly to create an instance of a polymorphic type. Thus, *number -> number* is an instance of α -> α, and so is *string -> string*.

Note that *number -> string* is not an instance of α -> α , since one α was replaced by a number and the other by a string (not uniformly). A polymorphic type can contain more than one type identifier, for example, α -> β. In this case, each identifier can be replaced separately. Thus, *number ->* β, α *-> string*, *number -> string*, *number -> number*, and γ *-> string* are all instances of the polymorphic type α -> β.

Note In deference to the typographically-challenged ASCII character set, the type assigned to nada is actually written 'a -> 'a. In typeset ActiveVRML documents, including this one, Greek letters are often used instead of the ASCII syntax for type identifiers in order to improve readability.

Every expression and declaration in ActiveVRML is assigned a type by the system using a standard Milner-Damas polymorphic type checker. Except to improve exposition and occasionally to resolve ambiguity with an overloaded operator, it is not necessary for the programmer to explicitly give types. An expression may be qualified with a type using the syntax like the following:

```
expression: type-expression
```

For example, the following syntax can be used to restrict *nada* to a particular type (desirable for clarity):

```
nada(val: string) = val
```

This will assign *nada* the *monomorphic* type *string -> string*.

The following sections define the basic types for ActiveVRML and list the constructor forms and functions for these types. Later sections define types for reactivity and for modeling (geometry, images, and associated types).

Unit Type: unit

Type **unit**

The **unit** type is a trivial type containing only one member. This type is often used for functions that take or return uninteresting data, similar to the way that the *void* type is used in C++ programs.

Constructors ()

The unique member of the unit type, pronounced "trivial."

Function Type: type -> type

Type **type -> type**

The function type α -> β represents mappings from type α to type β. Functions in ActiveVRML may be *higher-order*, meaning that a function can take another function as an argument or return a function as its result. For example, function *f* might have type (*number -> number*) *-> number*. This means that *f* can be applied to a function with type *number -> number* to produce a result of type *number*. Another function *g* might have type *number->*(*number -> number*). This means that *g* will take a number as an argument, and produce a function with type *number -> number* as its result.

Constructors **function pattern . expression**

This constructor is used to create anonymous function values. The pattern part of this constructor (described in the "Pattern Matching" section) may be thought of as a list for formal arguments. For example, the declaration

```
f (x, y) = x * y + 1
```

can be thought of as an abbreviation for:

```
f = function (x, y). x * y + 1
```

Function declarations are value declarations where the value is a function value.

Functions

infix o: (α -> β) * (β -> γ) -> (α -> γ)

The expression *f o g* is the *composition* of the functions *f* and *g*. The notation **infix o** means that **o** is an infix operator (like the familiar **+** as in 14 **+** 3). The value of (*f o g*)(*x*) is *f*(*g*(*x*)). Note that **o** has a type like a higher-order function. It takes two functions as arguments and returns a function as its result. Its type can be written as ((α -> β) * (β -> γ)) -> (α -> γ) since * has higher precedence than -> in ActiveVRML types. (See ActiveVRML Type Precedence later in this document.)

Product Type: type * type

Type

type * type

The product type α * β represents pairs of elements: for example, (*e1*, *e2*) where *e1* has type α and *e2* has type β.

Constructors

expression, **expression**

The pairing constructor is a comma. The precedence of the comma is extremely low in ActiveVRML, so it is usually desirable (and visually clearer) to write pairing with parentheses: (*3*, *"hello"*).

The pairing operator associates to the right. Thus, (*3*, *"hello"*, *17*) is the same as (*3*, (*"hello"*, *17*)). It is a pair, the second element of which is also a pair.

Functions

first: α * β -> α

first(α, β) returns the first element of a pair, α.

second: α * β -> β

second(α , β) returns the second element of a pair, β.

List Type: list

Type

α **list**

The type α **list** is a list (or finite sequence). Each element is of type α . For example, *number list* is a list of numbers, and (*string list*) *list* is a list where each element is a list of strings.

Constructors

[expression-list]

A list of expressions (zero or more) separated by commas. For example, *[]* is the null list (of type α *list*) and *[1, 2, 3]* is a number list.

Functions

head: α *list -> α*

head(*list*) returns the first element of the *list* list. It is illegal to apply **head** to an empty list.

tail: α *list -> α list*

tail(*list*) returns a list comprising all but the first element of the original list. It is illegal to apply **tail** to an empty list.

infix::: α * α *list -> α list*

The operator **:: i**s read as "cons." The expression *elt***:: l**is*t* returns a new list formed by prepending ("cons'ing") *elt* to *list*.

empty: α *list -> boolean*

empty(*list*) is true if and only if *list* is an empty list.

length: α *list -> number*

length(*list*) returns the length of *list*.

map: (α *->* β) * (α *list*) *->* β*list*

map(*fun*, *list*) returns a list by applying *fun* to each element of *list*.

reduce: (α * β *->* β) * β * (α list) -> β

reduce(*[e1,...,en]*, *base*, *fun*) returns:

```
fun(e1, fun(e2, fun(...,fun(en-1, fun(en, base))...)))
```

nth: α *list * number ->* α

nth(*list*, *n*) returns the *n*th element of *list*, where the first element of the list is 1.

Boolean Type: boolean

Type

boolean

The **boolean** type represents true and false values in ActiveVRML.

Constructors

true

Constructs a true value.

false

Constructs a false value.

Functions

infix and: *boolean * boolean -> boolean*

The expression *x* **and** *y* is true if *x* and *y* are true.

infix or: *boolean * boolean -> boolean*

The expression *x* **or** *y* is true if either *x* or *y* are true.

not: *boolean -> boolean*

The expression **not** *x* is true if *x* is false.

infix =: $\alpha * \alpha$ *-> boolean*

Equality may be used to compare any two values with the same type. Equality in ActiveVRML is structural: pairs are equal if each side of the pair is equal; lists are equal if their lengths are the same and corresponding elements of each list are equal. Equality applied to functions and modeling types is not defined (since it is not always possible to determine equality on these types).

infix <>: $\alpha * \alpha$ *-> boolean*

The expression *x* <> *y* is true if *x* is not equal to *y*.

Number Type: number

Type

number

The **number** type in ActiveVRML does not distinguish between "fixed point" and "floating point" numbers; both are considered numbers. The implementation will choose an appropriate representation.

Constructors

number-literal

The **number-literal** constructor is any sequence of characters satisfying the following regular expression:

```
digit+ ('.' digit*)? (['e' 'E'] ['+' '-']?digit+)?
```

time

A time-varying number representing the local time of a behavior. This important constructor is the basis for many interesting time-varying behaviors.

random

A pseudo-random number in *[0, 1]* that is time-varying. All instances of **random** that start at the same global time have the same time-varying value.

pi

A constant number representing π.

Functions

infix +: *number * number -> number*

The expression *x* **+** *y* returns the value of *x* added to *y*.

infix *: *number * number -> number*

The expression *x* *** *y* returns the value of *x* multiplied to *y*.

infix -: *number * number -> number*

The expression *x* **-** *y* returns the value of *y* subtracted from *x*.

infix /: *number * number -> number*

The expression *x* /*y* returns the value of *x* divided by *y*. Division by zero is an error.

prefix -: *number -> number*

The expression **-**x returns the value of *x* multiplied by *-1*.

prefix +: *number -> number*

The prefix **+** operator does not change the value of a number. Does this mean that, if x = -100, then +x = -100?

infix <: *number * number -> boolean*

The expression *x* < *y* returns true if the value of *x* is less than the value of *y*.

infix <=: *number * number -> boolean*

The expression *x* **<=** *y* returns true if the value of *x* is less than or equal to the value of *y*.

infix >: *number * number -> boolean*

The expression *x* >*y* returns true if the value of *x* is greater than the value of *y*.

infix >=: *number * number -> boolean*

The expression *x* >=*y* returns true if the value of *x* is greater than or equal to the value of *y*.

abs: *number -> number*

abs(*x*) returns the absolute value of *x*.

sqrt: *number -> number*

sqrt(*x*) returns the square root of *x*.

mod: *number * number -> number*

mod(*x*, *y*) returns the modulus of *x* divided by *y*.

ceiling: *number -> number*

ceiling(*x*) returns the smallest integer that is greater than or equal to *x*.

floor: *number -> number*

floor(*x*) returns the largest integer that is less than or equal to *x*.

round: *number -> number*

round(*x*) returns the nearest integer to *x*.

radiansToDegrees: *number -> number*

radiansToDegrees(*x*) returns *x* expressed in degrees.

degreesToRadians: *number -> number*

degreesToRadians(*x*) returns *x* expressed in radians.

asin: *number -> number*

asin(*x*) returns the arcsine of *x*.

acos: *number -> number*

acos(*x*) returns the arccosine of *x*.

atan: *number * number -> number*

atan(*h*, *w*) returns the arctangent of *h divided by w* in radians.

atan: *number -> number*

atan(*x*) returns the arctangent of *x*.

cos: *number -> number*

cos(*x*) returns the cosine of *x* in radians.

sin: *number -> number*

sin(*x*) returns the sine of *x* in radians.

tan: *number -> number*

tan(*x*) returns the tangent of *x* in radians.

infix ^: *number * number -> number*

The expression *x* **^** *y* returns *x* raised to the power of *y*.

exp: *number -> number*

exp(*x*) returns the exponential value of *x*.

ln: *number -> number*

ln(*x*) returns the natural logarithm of *x*.

log10: *number -> number*

log10(*x*) returns the base 10 logarithm of *x*.

seededRandom: *number -> number*

Pseudorandom behavior is parameterized by a random seed. **SeededRandom** returns *x* in *[0, 1]*, implicitly parameterized by time.

cubicBSpline: *number list * number list -> (number -> number)*

cubicBSpline(*knots, control points*) creates a polynomial, real-valued B-spline function of degree three.

nurb: *number list * number list * number list -> (number -> number)*

nurb(*knots, control points, weights*) creates a rational, real-valued B-spline function of degree three.

Reactive Behaviors

Recall that all values in ActiveVRML are potentially time-varying. Variation is achieved by specifying the time explicitly (for example, *3 + time*), input such as mouse motion, or by reactivity. This section defines reactivity and the constructs used to build reactive behaviors.

Reactivity

A reactive behavior is one that can potentially react to an event. The following is a very simple reactive behavior:

```
red until leftButtonPress => green
```

In this expression, **UNTIL** is the main operator. The expression is parsed into the following:

```
red until (leftButtonPress => green)
```

The reactive behavior changes the color from red to green when the mouse button is pressed.

The subexpression *leftButtonPress => green* is called a handler. It pairs up an event, *leftButtonPress*, with a value, *green*. The value is the action taken when the event occurs.

The **UNTIL** construct can also be used to watch for more than one event, reacting to the first one that occurs. For example:

```
red until leftButtonPress => green | rightButtonPress => yellow
```

This is parsed into the following:

```
red until ( (leftButtonPress => green)
        | (rightButtonPress => yellow) )
```

The color remains red until either the left or right mouse buttons are pressed. If the left button is pressed, the color changes to green. If the right button is pressed, the color changes to yellow.

In general, the logic of the **UNTIL** operator follows this pattern:

```
b0 until e1 => b1
     | e2 => b2
     .
     .
     .
     | en => bn
```

The reactive behavior is *b0* until any one of the events occurs, *e1* through *en*. The first event to occur, *ei*, results in the behavior, *bi*.

A more advanced form of events uses event data to produce the next behavior. For example:

```
0 until numEvent => function x. x + 1
```

In this case, *numEvent* is an event that produces a number. The value of this behavior is zero until the event occurs, and then becomes the value associated with the event plus 1.

The type checking of an **UNTIL** expression is as follows: If *b* is an α behavior and *e* is an α event, then *b until e* is a reactive behavior with type α behavior.

The next section describes events in more detail.

Events and Reactivity

An event can trigger a discrete change in a behavior. In addition to marking the occurrence of an event at a particular time, an event may also produce data. An α event produces some data of type α that can be used as new behavior after the event.

Events are constructed from one of the following types of events.

System Events

System events represent user input events. All of the following are system events:

- LeftButtonPress: *unit event*
- RightButtonPress: *unit event*
- KeyPress: *character event*

Boolean Events

Events corresponding to a boolean behavior occur as soon as the boolean value is evaluated as true. The predicate function is used to construct an event from a boolean behavior. For example, *predicate*($x = 0$) is a unit event (one that returns a unit as its data) that occurs the first time that behavior *x* is zero.

Simple Handler Events

New events may be constructed from other events. If *e* is a unit event (an event that does not produce interesting data) and *b* is an α behavior, then *e=>b* is a simple handler. In this example, the behavior is to wait for event *e* and before becoming *b*.

Handler Events

More complex handler events use the data produced by an event to construct the resulting behavior. If *e* is an α event and *f* is a function with type $\alpha \to \beta$, then *e=>f* is a β event. In this example, the behavior is to wait for event *e*, and then use the data produced by the event to run function *f*. The result is an event, *e=>f*, that occurs at the same time that *e* does and produces *f*(*x*) as the data, where *x* is the data produces by event *e*.

Alternative Events

If *e* and *e'* are α events, then *e* | *e'* is an α event. This means to choose whichever event, *e* or *e'*, happens first, and then return the corresponding data for the event. If *e* and *e'* happen at the same time, it is undefined which will be chosen.

Filtered Events

If *e* is an α event, and *p* is a function that maps α values to boolean values, then *suchThat*(*e*, *p*) is an α event that allows through only occurrences of *e* whose data satisfies the predicate *p*.

Snapshot Events

The snapshot construct may be used to make a time varying behavior into a static one (i.e., no longer varying with time). This is useful when one wishes to record the instantaneous value of a time varying behavior for some other use. For example,

```
0 until snapshot(xComponent(mousePosition), leftButtonPress,)
```

is the static behavior 0 until the left mouse button is pressed, and then becomes the static value of the x coordinate of the mouse position at the time of the press. The behavior

```
0 until leftButtonPress => xComponent(mousePosition)
```

is different in that it produces a behavior that continues to vary with time and track the x coordinate of the mouse after the button event occurs. The following subsections define the types and operators used in constructing reactive values.

Behavior Termination

Behaviors can terminate. The end construct is of type α (any type) and means to immediately terminate the behavior. For example:

```
b = time until leftButtonPress => end
```

This behavior varies with time until the *leftButtonPress* event occurs. Then it terminates.

A behavior will also terminate if one of its defining behaviors terminates. For example, consider

```
b' = f(b, 3)
```

If behavior *b* terminates, then *b'* terminates at the same time.

The **DONE** construct is a unit event that can be used to detect when a behavior has terminated and react accordingly. Consider the following:

```
repeat(b) = b until done => repeat(b)
```

This function takes behavior *b* and runs it until it terminates. Then it starts *b* again. (Note: This a built-in function in ActiveVRML.)

Does this last note mean that the example is a built-in function or that there is another function available that does the same thing?

Behaviors and Time

To improve modularity, all behaviors are defined to begin at local time zero. That is, when a behavior begins, no matter what the system time is, the time for the behavior starts at zero. Consider the following:

```
b until e => b'
```

When *b* is performed, there are two times to consider, the system time and the local time for *b*. Let the system time be *tg*. Whatever the value of *tg* is, at the time *b* is performed, it represents time zero for *b*. Event *e* occurs some time after *tg*. Let this time be *te*. This behavior can then be broken down by time: Do behavior *b* from time *tg* to *te*. Then do behavior *b'*. The local time for *b* is zero to (*te -tg*). When *b'* starts, its local time is zero as well.

Here is a simple behavior that uses the local time as its events:

```
green until time = 2 => (blue until time = 1 => red)
```

This behavior makes the color green for 2 seconds, blue for 1 second, and then red.

The TimeTransform Construct

The **timeTransform** construct can be used to change the interpretation of local time. The function has the type **timeTransform**: α * *number* -> α . It uses the number to redefine how local time is interpreted within a behavior. Consider the following:

```
doubleSpeed = time * 2;
b = playVideo(video);

doubleVideo = timeTransform(b, doubleSpeed)
```

In this example, the video is played at twice its original speed. From the perspective of global time, each 1 second interval corresponds to 2 seconds of local time. The effects of time transformation are cumulative. For example:

```
timeTransform(doubleVideo, doubleSpeed)
```

This line would play the video at four times its original speed.

To be consistent and predictable, the number argument (*n*) for the time transformation must satisfy two rules:

- Monotone—For all times *t0* and *t1* when *t0* is less than *t1*, *n* at time *t0* must be less than *n* at time *t1*.
- Nonnegative—For all times *t*, *n* at time *t* is nonnegative.

Monotonicity is required to make event reaction sensible (event transitions cannot be undone). Nonnegativity is required to prevent definition of a behavior before local time zero; that is, it may not make sense to sample a behavior like a video before local zero.

Certain behaviors, principally those defined by system or user input devices, may not be transformed in time in the same way that artificial behaviors can be. Such devices ignore user-defined time transformations when they are sampled.

Reactive Types

Following are definitions for the basic types for events, reactivity, and time.

Event Type: event

Type

α **event**

Constructors

done: *unit event*

The **done** constructor detects the termination of a behavior.

Functions

infix |: *α event * α event -> α event*

The expression *e1* **|** *e2* is an alternative event. The first of the two events is chosen, and the data associated with that event becomes the data for the alternative event.

predicate: *boolean -> unit event*

predicate(*b*) turns a boolean value into an event with trivial (unit) data. The event occurs the first time after local time 0 that the predicate *b* is true.

infix =>: *α event * (α -> β) -> βevent*

The expression *e* **=>** *h* is a handler event. It occurs the same time that *e* does, and returns as the function *h* applied to the data produced by *e*.

infix =>: *α event * β -> βevent*

This second form of *e* => *b* is a syntactic convenience, valid only when *b* is not a function. It is roughly equivalent to *e* => *function x.b* and is useful when the handler does not need the value of the data produced by the event. This is a special form and does not immediately evaluate the second argument.

suchThat: α *event* * (α -> *boolean*) -> α *event*

suchThat(*e*, *p*) is a filter event that occurs when *e* does, producing the data that *e* would, but only if the predicate *p* is true on that data.

andEvent: α *event* * β*event -> (*α * β*)event*

andEvent(*e1*, *e2*) occurs when *e1* and *e2* occur simultaneously. The data returned is the pair of data from *e1* and *e2*.

snapshot: α * *unit event* -> α *event*

snapshot(*b*, *e*) creates a new event that happens at the same time as the *e* event, and associates a static snapshot of the *b* behavior with the event. When the *e* event occurs, *b* is sampled. A new event with the static sampled value of *b* associated with the *e* event is created. Snapshot is a special form that does not immediately evaluate the second argument.

Reactivity Type

Constructors

end : α

The **end** constructor causes the behavior to finish immediately.

infix until: α * α *event* -> α

The expression *b* **until** *e* is equivalent to the behavior is *b* until the event *e* occurs, in which case, the behavior becomes *b'*.

repeat: α -> α

repeat(*b*) is equal to *b* until *b* ends. Then it restarts with *b* at that time.

Time Type

Functions

timeTransform: α * *number* -> α

timeTransform(*b*, *timeTrans*) adjusts the local time line for behavior *b* to follow the (time-varying) number *timeTrans*. For example, **timeTransform**(*b*,*time* * *2*) makes a behavior that runs twice as fast as *b* normally would. See the "Behaviors and Time" section above for more information.

Modeling Types

This section defines a broad range of types and associated functions that are useful for creating and manipulating ActiveVRML media values.

ActiveVRML uses the following conventions:

- Time is specified in seconds.
- Angles are specified in radians.
- Distances are specified in meters.
- Coordinate system conventions:
- The three-dimensional coordinate system used is a right-handed system with positive *X* to the right, positive *Y* up, and negative *Z* into the screen.
- Canonical 3D objects that are used (lights, cameras, microphones) are all positioned at the origin, facing -*Z* with +*Y* up.
- The two-dimensional coordinate system used has positive *X* to the right and positive *Y* up.
- The window in which images are viewed has its center at (0,0).

2D Point Type: point2

Type

point2

A two dimensional point.

Constructors

origin2: *point2*

Functions

point2Xy: *number * number -> point2*

point2Xy(*x*, *y*) takes two Cartesian coordinates, *x* and *y*, and returns a **point2** type.

point2Polar: *number * number -> point2*

point2Polar(*theta*, *rho*) takes two polar coordinates, *theta* and *rho*, and returns a **point2** type.

addVector: *point2 * vector2 -> point2*

addVector(*point*, *vector*) adds *vector* to *point* and returns a **point2** type.

subtractVector: *point2 * vector2 -> point2*

subtractVector(*point*, *vector*) subtracts *vector* from *point* and returns a **point2** type.

infix -: *point2 * point2 -> vector2*

The expression *p1* **-** *p2* creates a vector from *p1* to *p2*.

distance: *point2 * point2 -> number*

distance(*p1*, *p2*) returns the distance between *p1* and *p2* by finding the square root of the *x* and *y* coordinates of *p1* and *p2* squared.

distanceSquared: *point2 * point2 -> number*

distanceSquared(*p1*, *p2*) returns only the *x* and *y* coordinates of *p1* and *p2* squared. Use **distanceSquared** when you need to compare the distance between one set of points and another.

xComponent: *point2 -> number*

xComponent(*point*) returns the first coordinate of *point* (Cartesian).

yComponent: *point2 -> number*

yComponent(*point*) returns the second coordinate of *point* (Cartesian).

transformPoint2: *transform2 -> (point2 -> point2)*

transformPoint2(*transformation*) (*point*) returns a **point2** type by applying *transformation* to *point*.

thetaComponent: *point2 -> number*

thetaComponent(*point*) returns the theta component of *point* (polar).

phiComponent: *point2 -> number*

phiComponent(*point*) returns the phi component of *point* (polar)

2D Vector Type: vector2

Type

vector2

A two dimensional vector.

Constructors

xVector2: *vector2*

yVector2: *vector2*

zeroVector2: *vector2*

Functions

vector2Xy: *number * number -> vector2*

vector2Xy(*x*, *y*) constructs a **vector2** type from two Cartesian coordinates, *x* and *y*.

vector2Polar: *number * number -> vector2*

vector2Polar(*theta*, *rho*) constructs a **vector2** type from two polar coordinates, *theta* and *rho*.

normal: *vector2 -> vector2*

normal(*vector*) normalizes *vector* such that its length is equal to 1.

length: *vector2 -> number*

length(*vector*) returns the length of *vector* by finding the square root of the *x* and *y* coordinates squared.

lengthSquared: *vector2 -> number*

lengthSquared(*vector*) returns from *vector* only the *x* and *y* coordinates squared. Use **lengthSquared** when you need to compare the length of one vector to another.

infix +: *vector2 * vector2 -> vector2*

The expression *v1* **+** *v2* adds the two vectors to form a third vector.

infix -: *vector2 * vector2 -> vector2*

The expression *v1 - v2* subtracts *v2* from *v1* to form a third vector.

scaleVector2: *vector2 * number -> vector2*

scaleVector2(*vector*, *scale*) multiplies *vector* by *scale* and returns a **vector2** type.

dot: *vector2 * vector2 -> number*

dot(*v1*, *v2*) returns the dot product of the two vectors, *v1* and *v2*.

xComponent: *vector2 -> number*

xComponent(*vector*) returns the *x* component of *vector* (Cartesian).

yComponent: *vector2 -> number*

yComponent(*vector*) returns the *y* component of *vector* (Cartesian).

transformVector2: *transform2 -> (vector2 -> vector2)*

transformVector2(*transformation*) (*vector*) returns a **point2** type by applying *transformation* to *vector*.

thetaComponent: *vector2 -> number*

thetaComponent(*vector*) returns the *theta* component of *vector* (polar).

rhoComponent: *vector2 -> number*

rhoComponent(*vector*) returns the *rho* component of *vector* (polar).

2D Transformation Type: transform2

Type

transform2

2D transformations represent a mapping between 2D-space and 2D-space. They are used to transform various 2D objects, including **point2**, **vector2**, and **image**, all by the overloaded **apply** function listed in their relevant sections.

Constructors

identityTransform2: *transform2*

Creates a transformation that does not change the object it is applied to.

Functions

infix o: *transform2 * transform2 -> transform2*

The expression *t1* **o** *t2* composes two transformations into a single **transform2** type.

translate: *number * number -> transform2*

translate(*tx*, *ty*) moves an object by adding *tx* and *ty* to the object's *x* and *y* coordinates.

translate: *vector2 -> transform2*

translate(*vector*) moves an object by adding the *x* and *y* coordinates from *vector* to the object's *x* and *y* coordinates.

scale: *number * number -> transform2*

scale(*sx*, *sy*) scales an object by multiplying *sx* and *sy* by the object's *x* and *y* coordinates.

scale: *vector2 -> transform2*

scale(*vector*) scales an object by multiplying the *x* and *y* coordinates from *vector* by the object's *x* and *y* coordinates.

scale2: *number -> transform2*

scale(*number*) scales an object by multiplying *number* by the object's *x* and *y* coordinates.

rotate2: *number -> transform2*

rotate2(*number*) rotates an object *number* radians ccw. What is ccw ???

shear2: *number -> transform2*

shear2(*number*) shears an object in the *x* direction such that the object's x coordinate is increased by the product of its *y* coordinate multiplied by *number*.

transform3x2: *number * number * number * number * number * number -> transform2*

transform3x2 converts a matrix of two rows and three columns to two coordinates, *x* and *y*.

inverse: *transform2 -> transform2*

inverse(*trans*) creates a transformation that is inverse to *trans*.

isSingular: *transform2 -> boolean*

isSingular(*trans*) returns true if *trans* does not contain an inverse transformatin.

Image Type: image

Type

image

A value of type **image** is a spatially continuous image behavior of infinite spatial extent. Operations on it include the application of 2D transforms and opacity, and overlaying images. Continuous images are constructed by importing bitmap files, projecting 3D geometry, or by rendering text into an image.

Constructors

emptyImage: *image*

import(pathname.[bmp | jpeg | gif]): *image * vector2 * number*

Import a bitmap file. The return value *image* is the imported image, centered at (0,0). *point2* is the upper-righthand coordinate of the resultant image, and *number* is the resolution of the image in pixels per meter.

Functions

renderedImage: *geometry * camera -> image*

The **renderedImage**(*geometry*, *viewingCamera*) function is the primary 3D to 2D interface. The *viewingCamera* parameter determines the projection by which the *geometry* will be imaged. The resultant image is spatially infinite, and the camera performs no cropping. Cropping, if desired, can be achieved by using the *crop* function below. It may be useful to think of this function as returning the entire projection plane as an image. The description of the **camera** type discusses the projection plane and projection point.

infix over: *image * image -> image*

The expression *top* **over** *bottom* constructs a new image by placing the *top* image over the *bottom* image.

Is this a continuation of renderedImage definiation above or a typo?

opacity2: *number* -> (*image -> image*)

opacity2(*value*)(*img*), given a value from 0.0 to 1.0, returns a new image identical to *img*, but with a certain percentage of opacity, determined by *value*: *percentage opacity = value * 100*. This function composes multiplicatively; thus **opacity2**(*0.5*)(*opacity2*(*0.2*)(*myOpaqueImg*)) results in an image with opacity of 0.1 (or 90% transparent).

crop: *point2 * point2* -> (*image -> image*)

crop(*min*, *max*)(*img*) returns an image identical to *img* inside of the box defined by *min* and *max*, and a transparent layer outside of this box.

tile: *point2 * point2* -> (*image -> image*)

tile(*min*, *max*)(*img*) returns an infinitely tiled image given by cropping *img* to the region (*min*,*max*), and then replicating that region infinitely in all directions.

transformImage: *transform2 -> (image -> image)*

transformImage(*transformation*) (*image*) returns an **image** type by applying *transformation* to *image*.

Composite 2.5D Image Type: montage

Type **montage**

A **montage** is a set of images with associated depth values. Montages are useful for creating multilayered, image-based (cel) animation.

Constructors **emptyMontage**

Constructs a **montage** object with no images.

Functions **imageMontage**: *image * number -> montage*

imageMontage(*image*, *depth*) builds a 2.5D image set with a single image at *depth*.

infix union: *montage * montage -> montage*

The expression *m1* **union** *m2* combines the contents of two image sets, *m1* and *m2*, into a single collection.

renderedImage: *montage -> image*

renderedMontage(*montage*) converts the set of images and depths encapsulated in *montage* into a flattened image. Image elements with larger depths will be layered underneath.

3D Point Type: point3

Type **point3**

A three dimensional point.

Constructors **origin3**

Functions **point3Xyz**: *number * number * number -> point3*

point3Xyz(*x*, *y*, *z*) constructs a **point3** type from three Cartesian coordinates.

point3Spherical: *number * number * number -> point3*

point3Spherical(*theta*, *phi*, *rho*) constructs a **point3** type from three polar coordinates.

infix -: *point3 * point3 -> vector3*

The expression *p1* **-** *p2* creates a vector from *p1* to *p2*.

distance: *point3 * point3 -> number*

distance(*p1*, *p2*) returns the distance between *p1* and *p2* by finding the square root of the *x*, *y*, and *z* coordinates of *p1* and *p2* squared.

distanceSquared: *point3 * point3 -> number*

distanceSquared(*p1*, *p2*) returns only the *x*, *y*, and *z* coordinates of *p1* and *p2* squared. Use **distanceSquared** when you need to compare the distance between one set of points and another.

addVector: *point3 * vector3 -> point3*

addVector(*point*, *vector*) adds *vector* to *point* to return a new **point3** type.

subtractVector: *point3 * vector3 -> point3*

subtractVector(*point*, *vector*) subtracts *vector* to *point* to return a new **point3** type.

transformPoint3: *transform3 -> (point3 -> point3)*

transformPoint3(*transformation*) (*point*) returns a **point3** type by applying *transformation* to *point*.

xComponent: *point3 -> number*

xComponent(*point*) returns the *x* component of *point* (Cartesian).

yComponent: *point3 -> number*

yComponent(*point*) returns the *y* component of *point* (Cartesian).

zComponent: *point3 -> number*

zComponent(*point*) returns the *z* component of *point* (Cartesian).

thetaComponent: *point3 -> number*

thetaComponent(*point*) returns the *theta* component of *point* (Polar).

phiComponent: *point3 -> number*

phiComponent(*point*) returns the *phi* component of *point* (Polar).

rhoComponent: *point3 -> number*

rhoComponent(*point*) returns the *rho* component of *point* (Polar).

3D Vector Type: vector3

Type

vector3

A three dimensional vector. Direction and magnitude. ???

Constructors

xVector3: *vector3*

yVector3: *vector3*

zVector3: *vector3*

zeroVector3: *vector3*

Functions

vector3Xyz: *number * number * number -> vector3*

> **vector3Xyz**(*x*, *y*, *z*) constructs a **vector3** type from three Cartesian coordinates, *x*, *y*, and *z*.

vector3Spherical: *number * number * number -> vector3*

> **vector3Spherical**(*theta*, *phi*, *rho*) constructs a **vector3** type from three polar coordinates, *theta*, *phi*, and *rho*.

normal: *vector3 -> vector3*

> **normal**(*vector*) normalizes *vector* such that its length is equal to 1.

length: *vector3 -> number*

> **length**(*vector*) returns the length of *vector* by finding the square root of the *x*, *y*, and *z* coordinates squared.

lengthSquared: *vector3 -> number*

> **lengthSquared**(*vector*) returns from *vector* only the *x*, *y*, and *z* coordinates squared. Use **lengthSquared** when you need to compare the length of one vector to another.

infix +: *vector3 * vector3 -> vector3*

> The expression *v1* **+** *v2* adds the two vectors to form a third vector.

infix -: *vector3 * vector3 -> vector3*

> The expression *v1* - *v2* subtracts *v2* from *v1* to form a third vector.

scaleVector3: *vector3 * number -> vector3*

> **scaleVector3**(*vector*, *scale*) multiplies *vector* by *scale* and returns a **vector3** type.

dot: *vector3 * vector3 -> number*

> **dot**(*v1*, *v2*) returns the dot product of the two vectors, *v1* and *v2*.

cross: *vector3 * vector3 -> vector3*

> **dot**(*v1*, *v2*) returns the cross product of the two vectors, *v1* and *v2*.

transformVector3: *transform3 -> (vector3 -> vector3)*

> **transformVector3**(*transformation*) (*vector*) returns a **vector3** type by applying *transformation* to *vector*.

xComponent: *vector3 -> number*

> **xComponent**(*vector*) returns the *x* component of *vector* (Cartesian).

yComponent: *vector3 -> number*

> **yComponent**(*vector*) returns the *y* component of *vector* (Cartesian).

zComponent: *vector3 -> number*

> **zComponent**(*vector*) returns the *z* component of *vector* (Cartesian).

thetaComponent: *vector3 -> number*

thetaComponent(*vector*) returns the *theta* component of *vector* (polar).

phiComponent: *vector3 -> number*

phiComponent(*vector*) returns the *phi* component of *vector* (polar).

rhoComponent: *vector3 -> number*

rhoComponent(*vector*) returns the *rho* component of *vector* (polar).

3D Transformation Type: transform3

Type

transform3

3D transformations represent a mapping between 3D-space and 3D-space and are used to transform various 3D entities, including **point3**, **vector3**, **geometry**, **microphone**, and **camera**, all by using the overloaded **apply** function listed in their relevant sections.

Constructors

identityTransform3

Creates a transformation that does not change the object it is applied to.

Functions

infix o: *transform3 * transform3 -> transform3*

The expression *t1* **o** *t2* composes two transformations into a single **transform3** type.

translate: *number * number * number -> transform3*

translate(*tx*, *ty*, *tz*) moves an object by adding *tx*, *ty*, and *tz* to the object's *x*, *y*, and *z* coordinates.

translate: *vector3 -> transform3*

translate(*vector*) moves an object by adding the *x*, *y*, and *z* coordinates from *vector* to the object's *x*, *y*, and *z* coordinates.

scale: *number * number * number -> transform3*

scale(*sx*, *sy*, *sz*) scales an object by multiplying *sx*, *sy*, and *sz* by the object's *x*, *y*, and *z* coordinates.

scale: *vector3 -> transform3*

scale(*vector*) scales an object by multiplying the *x*, *y*, and *z* coordinates from *vector* by the object's *x*, *y*, and *z* coordinates.

scale3: *number -> transform3*

scale(*number*) scales an object by multiplying *number* by the object's *x*, *y*, and *z* coordinates.

rotate: *vector3 * number -> transform3*

rotate(*vector*, *number*) rotates an object *number* radians about the axis specified by *vector*.

xyShear: *number -> transform3*

xyShear2(*number*) shears an object in the *x* direction such that the object's x coordinate is increased by the product of its *y* coordinate multiplied by *number*.

yzShear: *number -> transform3*

yzShear2(*number*) shears an object in the *y* direction such that the object's y coordinate is increased by the product of its *z* coordinate multiplied by *number*.

zxShear: *number -> transform3*

zxShear2(*number*) shears an object in the *z* direction such that the object's z coordinate is increased by the product of its *x* coordinate multiplied by *number*.

transform4x4: *number * number * number * number * number * number * number * number * number * number * number * number * number * number * number * number -> transform3*

transform4x4 converts a matrix of four rows and four columns to four coordinates, *x*, *y*, *z*, and *w*.

lookAtFrom: *point3 * point3 * vector3 -> transform3*

lookAtFrom(*from*, *to*, *up*) creates a transformation when applied to an object centered at the origin, with *+Y* up and directed toward *-Z*, moves the object to *from*, pointing towards *to*, with its up direction as close to *up* as possible.

inverse: *transform3 -> transform3*

inverse(*trans*) creates a transformation that is the inverse of *trans*.

isSingular: *transform3 -> boolean*

isSingular(*trans*) returns true if *trans* does not have an inverse transformatin.

3D Geometry Type: geometry

Type

geometry

A value of type **geometry** is a spatially continuous behavior of infinite spatial extent in three dimensions. A geometry value is constructed by importing other geometry formats (such as VRML), applying modeling transformations, material properties, aggregating multiple geometries, positioning sound in 3D, and specifying other attributes.

Constructors

emptyGeometry: *geometry*

import(filename.[wrl]): *geometry * point3 * point3*

In the **import** constructor, **geometry** is the result of importing the specified file. The two points returned are the minimum and maximum extents of the tightest axis-aligned, rectangular bounding volume containing the geometry.

import(beginLiteral wrl ascii *body* endLiteral): *geometry * point3 * point3*

An alternative **import** constructor, allows VRML 1.0 geometry to be expressed in line. *body* is a sequence of ascii characters in VRML 1.0 format.

Functions

infix union: *geometry * geometry -> geometry*

g1 union g2 aggregates two geometries into their geometric union.

soundSource3: *sound -> geometry*

The **soundSource3** function allows sounds to be embedded into a geometry. It creates a geometry with the specified sound positioned at the local coordinate system origin. The resultant geometry may be transformed in space, and has no visible presence when rendered. The function **renderedSound**, described in the Sound section below, takes a **geometry** and a **microphone**, and creates a sound by spatializing all of the sounds embedded into that geometry with respect to the microphone.

transformGeometry: *transform3 -> (geometry -> geometry)*

transformGeometry(*transformation*) (*geometry*) returns a **geometry** type by applying *transformation* to *geometry*.

opacity3: *number ->* (*geometry -> geometry*)

opacity3(*value*)(*geo*), given a value from 0.0 to 1.0, returns a new geometry identical to *geo*, but with a certain percentage of opacity, determined by *value*: *percentage opacity = value * 100*. This function composes multiplicatively, making **opacity3**(*0.5*)(*opacity3*(*0.2*)(*myOpaqueGeo*)) result in a geometry with opacity of 0.1 (or 90% transparent).

texture: *image ->* (*geometry -> geometry*)

The **texture** function is the means by which texture mapping onto geometry is specified. The coordinates of the image are mapped onto the texture map coordinates associated with the vertices of the primitive geometries comprising the geometry being mapped, resulting in textured geometry. If the primitive geometries have no texture coordinates, texturing is ignored. Note that textures are applied in an outer-overriding fashion. That is, **texture**(*im1*)(*texture*(*im2*)(*geo*)) results in *geo* textured with *im1*.

Note The following functions create *light* geometries, all of which have no visible appearance themselves; but, they do cast light onto other objects they are aggregated with. The *point* and *spot* lights are both located at the origin. The *directional* and *spot* lights both point along the -Z axis.

---

ambientLight: *geometry*

directionalLight: *geometry*

pointLight: *geometry*

spotLight: *number * number * number -> geometry*

The **spotLight** function has arguments *fullcone*, *cutoff*, and *exponent*.

lightColor: *color* -> (*geometry -> geometry*)

lightColor(*col*)(*geo*) creates a geometry identical to *geo* where all the lights are *col* color. By default, the light colors are white.

lightAttenuation: *number * number * number* -> (*geometry -> geometry*)

lightAttenuation(*c*, *l*, *q*)(*geo*) creates a geometry identical to *geo* where Lambertian light attenuation equation is set to $1 / (c + ld + qdd)$ where *d* is the distance from the light to the object. By default, the attenuation coefficients are (*1*, *0*, *0*).

Note The following functions allow for attributing geometry with standard Lambertian shading characteristics. The outermost applied attribute overrides other attributes of the same kind; that is, **diffuseColor**(*red*)(*diffuseColor*(*blue*)(*geo*)) results in a red geometry.

---

diffuseColor: *color* -> (*geometry -> geometry*)

ambientColor: *color* -> (*geometry -> geometry*)

specularColor: *color* -> (*geometry -> geometry*)

emissiveColor: *color* -> (*geometry -> geometry*)

specularExponent: *number* -> (*geometry -> geometry*)

Camera Type: camera

Type

camera

The camera type is used to project geometry into an image via the **renderedImage** function.

Constructors

defaultCamera : *camera*

The canonical camera, **defaultCamera**, has its projection point at (0,0,1), looking along the *-Z* axis with *+Y* up, with the projection plane in the *XY* plane at $z = 0$.

New cameras may be created by applying transformations to existing cameras in order to position and orient the camera, and also to affect projection properties. Translation and orientation position and orient the camera, respectively. Scaling in X or Y stretches the resulting projection accordingly. Scaling in *Z* changes the distance between the projection point and the projection plane. For instance, a *Z* scale of less than one yields a wide-angle camera, while a Z scale of ∞ will yield a parallel projecting camera. (In practice, scaling in Z by a very large number is sufficient.)

Functions

transformCamera: *transform3 -> (camera -> camera)*

transformCamera(*transformation*) (*camera*) returns a **camera** type by applying *transformation* to *camera*.

Sound Type: sound

Type

sound

A **sound** value is constructed by importing primitive sounds, mixing, rendering of geometry with embedded sounds, and by the application of audio attributes, such as gain.

Note that sound is always considered to be single channel. Stereo is supported by constructing two separate sounds.

Certain audio effects can be achieved by using the general time transformation mechanism. For example, both phase shift and rate control can be achieved by time transformations.

Constructors

silence

import(pathname.[wav | au | aiff]): *sound * sound * number*

Importing a .wav, .au, or .aiff file constructs a pair of sounds, one for the left and right channels of the sound. If the imported file is monophonic, the two returned sounds are identical. When a sound is finished, it terminates. Sounds can be looped by using the **repeat** facility. The third returned value, a number, is the length in seconds of the longer of the two returned channels.

Functions

infix mix: *sound * sound -> sound*

The expression *s1* **mix** s*2* combines two sounds, *s1* and *s2*, into a single sound, with each component sound contributing equally.

renderedSound: *geometry * microphone -> sound*

renderedSound(*geo*, *mic*) produces an audio rendering of the sounds embedded within geo (by using **soundSource3**), with respect to *mic*.

gain: *number ->* (*sound -> sound*)

gain(*value*) multiplicatively adjusts the gain of a sound. Thus, *gain*(*0.3*)(*gain*(*5.0*)(*origSound*)) results in a sound 1.5 times as loud as the original sound.

Microphone Type: microphone

Type **microphone**

The **microphone** type represents an audio perceiver in 3D. ActiveVRML 1.0 supports a simple microphone that may be spatially transformed by using modeling transformations. Future versions will add other attributes to the microphone.

Constructors **defaultMicrophone**

The canonical camera, **defaultMicrophone**, lies at the origin, looking along the *-Z* axis with *+Y* up.

Functions **transformMicrophone**: *transform3 -> (microphone -> microphone)*

transformMicrophone(*transformation*) (*mic*) returns an **microphone** type by applying *transformation* to *mic*.

Color Type: color

Type **color**

Constructors **red**: *color*

green: *color*

blue: *color*

cyan: *color*

magenta: *color*

yellow: *color*

white: *color*

black: *color*

Functions **colorRgb**: *number * number * number -> color*

colorRgb(*red*, *green*, *blue*) constructs a **color** type using the color values, *red*, *green*, and *blue*.

colorHsl: *number * number * number -> color*

colorHsl(*hue*, *saturation*, *lightness*) constructs a **color** type using the values, *hue*, *saturation*, and *lightness*.

redComponent: *color -> number*

redComponent(*color*) returns the red color value of *color*.

greenComponent: *color -> number*

greenComponent(*color*) returns the green color value of *color*.

blueComponent: *color -> number*

blueComponent(*color*) returns the blue color value of *color*.

hueComponent: *color -> number*

hueComponent(*color*) returns the hue value of *color*.

saturationComponent: *color -> number*

saturationComponent(*color*) returns the saturation value of *color*.

lightnessComponent: *color -> number*

lightnessComponent(*color*) returns the lightness value of *color*.

Character Type: char

Type

char

Constructors

'c'

In the **'c'** constructor, *c* is an ASCII character or one of the following escape forms:

Escape Code	**Result**
\n	Newline
\t	Tab
\'	Apostrophe
\"	Quote
\\	Backslash
\integer	The ASCII character with this value

Functions

ord: *char -> number*

ord(*c*) returns the ASCII code for character *c*.

chr: *number -> char*

chr(*n*) returns the ASCII character corresponding to *n*.

String Type: string

Type **string**

Constructors "*string-literal*"

In the "*string-literal*" constructor, *string-literal* is a sequence of characters or escape characters.

Functions **infix** &: *string * string -> string*

The expression *string1* & *string2* concatenates the two strings and returns the result.

implode: *char list -> string*

implode(*char list*) returns a **string** built from *char list*.

explode: *string -> char list*

explode(*string*) returns a **char list** built from *string*.

numberToString: *number * number -> string*

numberToString(*num*, *precision*) function formats *num* to a **string** with *precision* digits after the decimal point. If *precision* is zero, then the decimal point is elided. It is illegal for *precision* to be negative.

Font Family Type: fontFamily

Type **fontFamily**

Constructors **serifProportional**: *fontFamily*

sansSerifProportional: *fontFamily*

monospaced: *fontFamily*

Text Type: text

Type **text**

ActiveVRML supports the construction of simply formatted text, which can then be rendered into an image. ActiveVRML 1.0 supports simple scaling, coloring, emboldening, italicizing, and choosing from a fixed set of typefaces.

Functions

simpleText: *string -> text*

The text created by **simpleText** has a default color (black), family (serif proportional), and is neither bold nor italic. It has a nominal scale of 1 point.

textColor: *color* -> (*text -> text*)

textColor creates an attributor such that you can assign the color of the text.

textFamily: *fontFamily* -> (*text -> text*)

textFamily selects the text font.

This could use some explanation.

bold: *text -> text*

bold(*text*) returns a new **text** type with emboldened text.

italic: *text -> text*

italic(*text*) returns a new **text** type with italicized text.

renderedImage: *text -> image * point2*

The **renderedImage** function takes **text** and returns an **image**, with the text centered around (0,0) in the image. The returned **point2** is the coordinate of the upper-right hand corner of the the nontransparent region of the resultant image. The resultant image may subsequently be scaled by applying image transformations.

The resultant image is transparent in all places other than where the text is actually rendered.

There are explicitly no alignment operators (align left, right, center, etc.), as these can be achieved by transformation of the resultant image.

Integration, Differentiation, and Interpolation

Derivatives, integrals, and linear interpolation apply to the following types:

Type	**Derivative**
number	number
point2	vector2
vector2	vector2
point3	vector3
vector3	vector3

```
derivative: T -> DT
integral: DT -> DT
```

Note that the derivative of a point is a vector, but the integral of a vector is a vector.

```
blendLinear: T * T * number -> T
```

Single User Interactivity

Interaction with a user is one way in which a behavior value can change over time. For ActiveVRML 1.0 models, a very simple user input model for a single user is provided. It provides behaviors that represent the user's mouse and keyboard, and facilities for specifying reactions to picking (clicking on) geometry and images.

User Input

The keyboard and mouse of the user are modeled through a number of additional, pervasive library functions and values in ActiveVRML:

leftButtonState: *boolean*

Tracks the state of the left button of the mouse.

leftButtonDown: *unit event*

Occurs each time the user presses the left mouse button.

leftButtonUp: *unit event*

Occurs each time the user releases the left mouse button.

rightButtonState: *boolean*

rightButtonDown: *unit event*

rightButtonUp: *unit event*

Analogous to the corresponding left button behaviors.

keyState: *character -> boolean*

For *keyDown*(*c*), the value is true if key *c* is presently depressed.

keyDown: *character event*

Occurs when a key is pressed. It produces the pressed character in the event data.

keyUp: *character event*

Occurs when a key is released. It produces the released character in the event data.

mousePosition: *point2*

Tracks the current position of the mouse in world image coordinates.

Picking Images and Geometry

ActiveVRML 1.0 provides a simple model of picking geometry and images within a model. Picking is based upon a continuously probing input device, the user's mouse. The following probe functions take images (for 2D) or geometry (for 3D) and return events that occur when any ray, cast from the assumed eye point of the viewer, "touches" the specified image or geometry.

Occlusion is taken into account. That is, probing rays do not pass through one nontransparent image and into another image, or through one nontransparent geometry into another.

The ActiveVRML functions for picking are:

```
pickable: image * (string list) -> image * (point2 * vector2) event
pickable: geometry * (string list) -> geometry * (point3 * vector3)
event
```

As an example of how these functions are used, consider:

```
origGeo = ...;
pickableGeo, pickEv = pickable(origGeo, ["my geo"]);
newGeo = pickableGeo until pickEv => function (pt,vec) . emptyGeometry;
```

(The discussions here are focused on geometry, but the identical properties hold for image picking.)

When the user's probe device (typically his mouse) is over pickableGeo, pickEv will fire causing the event transition, making the geometry invisible (by transitioning to the empty geometry). Somewhat more formally, in,

```
geo', ev = pickable(geo, path)
```

geo' behaves identically to geo, however, when the probe device is over it, the event ev is fired. The event data that ev is invoked with is the static point of intersection between the probe and the geometry (in the local coordinates of geo), and a vector-valued behavior that tracks the probe as it moves relative to the pick point (also in local coordinates).

The resultant geometry and event are completely insulated from any subsequent modeling transformations applied to the geometry. Thus, the geometry can be arbitrarily transformed (and multiply referred to), but the event will still fire when any instance, in any coordinate frame, is picked.

The path argument to pickable is a string list that is intended to allow the author to disambiguate between multiple instances of the same geometry or image in situations where different reactions are desired. For instance:

```
origGeo = ...;
pickableGeo, pickEv = pickable(origGeo, ["my geo"]);
geo1 = transformGeometry(xf1)(pickableGeo);
geo2 = transformGeometry(xf2)(pickableGeo);
reactive1 = geo1 until pickEv => function (pt,vec) . emptyGeometry;
reactive2 = geo2 until pickEv => function (pt,vec) . emptyGeometry;
model = reactive1 union reactive2;
```

In this scenario, if either reactive1 or reactive2 is picked, both change to emptyGeometry, since the same event is being used. If the author wants each instance to behave independently under picking, multiple invocations of pickable would be used, each with a different path argument:

```
origGeo = ...;
geo1, ev1 = pickable(origGeo, ["first"]);
geo2, ev2 = pickable(origGeo, ["second"]);

rg1 = transformGeometry(xf1)(geo1);
rg2 = transformGeometry(xf2)(geo2);

reactive1 = rg1 until ev1 => function (pt,vec) . emptyGeometry;
reactive2 = rg2 until ev2 => function (pt,vec) . emptyGeometry;
model = reactive1 union reactive2;
```

The path argument exists in order to allow different events to be returned from the same base geometry. Because the return values of all of the functions in ActiveVRML are dependent solely upon the value of the inputs, multiple calls to pickable with the same geometry and the same path will return the same geometry * event pair each time. Thus, the path argument exists to allow the author to explicitly disambiguate multiple instances, when desired.

Pattern Matching

This section describes a useful, somewhat more advanced feature of ActiveVRML: the ability to specify declarations using pattern matching. The general syntax for a value declaration is as follows:

```
pattern = expression
```

The general syntax for a function declaration is as follows:

```
identifier pattern = expression
```

Patterns may be used to destructure values and specify bindings for (potentially) more than one identifier. Patterns are denoted in one of the following forms:

()

Matches the trivial value, (), of the type unit.

identifier

Matches any value and effectively binds the value to the identifier in the right hand side of the declaration. All of the identifiers in a pattern must be distinct.

pattern1, *pattern2*

Matches a pair value if *pattern1* and *pattern2* match the left and right components of the pair. The comma pattern associates right.

(*pattern*)

Groups pattern syntax for readability and precedence. The form matches if *pattern* matches the value.

pattern: *type*

Matches the value if *pattern* does, and constrains the pattern to have a particular type.

The following are a few example declarations:

x = 3

This declares an identifier *x* as a variable with a value of 3.

(x, y) = (4, 17)

This pattern matches since the pattern (*x*, *y*) and value (*4*, *17*) are both pairs. Effectively, *x* is bound to *4*, and *y* is bound to *17*.

(x, y) = p

This declaration matches *x* to the first component of *p* (which must be a pair) and *y* to the second component of *p*.

(x, y) = 3

This declaration will cause an error to be reported since 3 is not a pair.

f() = 5

This declares a constant function. The application expression *f*() will always produce a value of 5.

g(x) = x + 1

This declares a function *g* that takes an argument *x*.

h(x, y) = x + y

This declares a function *h* that takes a single argument that must be a pair. The function may be applied either as *h*(*3*, *4*) or as *h*(*p*) where *p* is a pair of numbers.

k(x, y, z) = x + y / z

This declares a function k that takes a single argument that is a pair, the second component of the pair being also a pair. Because comma patterns associate right, the argument takes the following form:

```
(int, (int, int))
```

An application of *k* could look like *k*(*1*, *2*, *3*), *k*(*1*, (*2*, *3*)), *k*(*1*, *p*) where *p* is a pair, or *k*(*t*) where *t* is a triple of type *int*int*int*. The application *k*((*1*,*2*),*3*) will report an error.

Pattern matching can also be used to specify the formals for a function expression. The following is an anonymous addition function:

```
function (x, y). x + y
```

And, this function returns the first element of a pair:

```
function (x, y). x
```

ActiveVRML Models and World Wide Web Browsing

This section describes the conventions that must be followed to connect an ActiveVRML model to a World Wide Web (web) browser for single-user interactive animations.

ActiveVRML 1.0 models should contain the following comment as their first source line:

```
// ActiveVRML 1.0 ASCII
```

An ActiveVRML model consists of a list of top-level declarations.

An external entry point will be a declaration with type *geometry* or *image*sound*sound*. These are the only behaviors that may be indexed from a uniform resource locator (URL). The former will cause the browser to enable a 3D navigational user interface to allow the user to navigate the geometry, and have the embedded sounds rendered (played). The latter form allows the creator of the model to use their own camera, and to directly specify the sounds to play to the left and right speakers.

Embedding and Hyperlinking To ActiveVRML

To specify a URL from an HTML document to an ActiveVRML model, use the following syntax:

```
<a href="http://www.microsoft.com/model.av#mainEntry">
```

This will create an active link to the model *myModel* in the file *model.av* on the server *www.microsoft.com*. The entry point for the model, *mainEntry*, must be type *geometry* or *image*sound*sound*. The former is for geometries that will use the default camera from the view (including embedded sound). The latter is for images with stereo sound.

When the user clicks the URL, an ActiveVRML 1.0 capable viewer will begin executing the ActiveVRML (at local time 0).

It is also possible to embed an ActiveVRML model within an HTML page. To do so, use the following syntax:

```
<embed clsid=ActiveVRML.ActiveVRMLView1 width=300 height=300
props="URL=http://www.microsoft.com/model.av#mainEntry">
```

This instructs the viewer to display the model in a window within the HTML page. The height and width are specified in pixels to be consistent with other embedded types in HTML.

Hyperlinking from ActiveVRML

The hyperlinking interfaces in ActiveVRML are very basic:

```
hyperlink3: string -> (geometry -> geometry)
hyperlink2: string -> (image -> image)
```

These act as attributers for geometries and images. For example:

```
im2 = hyperlink2("http://www.microsoft.com")(im1)
```

The variable *im2* is now an image that when selected will be noticed by the browser, causing a jump to the specified URL. Note that the URL can be any valid web content type, not just ActiveVRML.

Viewer Conventions and Information

The ActiveVRML viewer window will obey a number of conventions that define its interaction with the model being displayed:

The origin of displayed images will be at the center of the viewer window.

When the window is resized, its coordinate range changes. Thus, generally, more or less of the model comes into view, rather than the model being stretched or squeezed. The model only reacts to changes in the size of the window if it is making use of the **viewerUpperRight** behavior described below.

The following information is available to the model in ActiveVRML:

viewerResolution: *number*

The resolution of the view in pixels per meter. In general, this number will be an approximation, as, among other things, monitor size varies.

viewerUpperRight: *point2*

The coordinate of the upper-right corner of the viewer window. This is useful for models which size or position objects explicitly with respect to the window the model is being viewed in.

ActiveVRML Grammar and Lexical Conventions

The following information describes ActiveVRML grammatical formations and lexical conventions.

Identifiers

An ActiveVRML identifier is an alphanumeric string beginning with a letter. Identifiers are case sensitive. No length limit is imposed.

Type identifiers

A type identifier is an apostrophe (') followed by an identifier. In typeset documents, including this one, Greek letters are often used instead of ASCII for type identifiers to improve readability.

Comments

Comments are of the form

```
/* This is a comment. */
// This is a comment.
```

where the first form encloses any characters up to the first */ pattern and the latter form ignores everything until the end of line. Nested comments are handled correctly.

White Space

Spaces, tabs, carriage-returns, line-feeds, and comments are considered white space and are ignored other than as token separators.

ActiveVRML Keywords

Following are the keywords of ActiveVRML. These words are reserved and may not be used as identifiers:

and

else

event

function

if

import

in
let
list
mix
not
o
or
over
then
union
until

ActiveVRML Precedence Table

Operator	**Associativity**
->	Right
* (product type)	Left
list event	Non Associative
, (comma)	Right
.(dot in function) else in	Non Associative
until	Right
| (or event)	Left
=> (event handler)	Left
o union over mix	Left
:: (list cons)	Right
or	Left
and	Left
not	Non Associative
= < <= > >= <>	Left
+ - (binary) &	Left
* /	Left
^	Right
+ - (unary)	Non Associative
: (type qualification)	Non Associative

ActiveVRML Syntax

```
program:
```

```
declarations
| epsilon

declarations:
declaration
| declaration ;
| declaration ; declarations

declaration:
pattern = commaexpression
| identifier pattern = commaexpression
pattern:
 ()
| identifier
| pattern , pattern
| (pattern)
| pattern: typeexp

expressionlist:
nonemptyexpressionlist
| epsilon
nonemptyexpressionlist:
expression , nonemptyexpressionlist
| expression

commaexpression:
expression , commaexpression
| expression
expression:
if expression then expression else expression
| let declarations in expression
| function pattern . expression
| expression binaryoperator expression
| unaryoperator expression
| applyterm

binaryoperator:
 + | - | * | / | ^ | = | > | >= | < | <= |:: | & | and | or | until | |
| union | over | mix | o

unaryoperator:
 + | - | not

applyterm:
applyterm term
| term
term:
 numberliteral
 | characterliteral
 | stringliteral
```

```
| identifier
| ()
| (commaexpression )
| [ expressionlist]
| term: typeexp

typeexp:
typeidentifier
| identifier
| typeexp * typeexp
| typeexp -> typeexp
| typeexp identifier
| (typeexp)

epsilon:
/* empty */
```