

Computer Graphics

An Introduction to VRML 2.0

Dr. Michael Laszlo

Nova Southeastern University

School of Computer and Information Sciences

2001

Outline

- The purpose of this set of slides is to cover the fundamental features of VRML 2.0. See the Reference section for more comprehensive treatments.
- Topics
 - Introduction
 - Key Concepts
 - Primitive Shapes
 - Transforming Shapes
 - Animating Worlds
 - Sensing Viewer Actions
 - Sample VRML Programs
 - References

Introduction: What is VRML?

- Virtual Reality Modeling Language (VRML) is a file format for describing objects. Most of the objects pertain to 3D virtual worlds: 3D geometries, materials, images, sounds, and so forth.
- The file format is textual, and is designed to be easy to read and write both by humans and by sundry software tools. It can serve as a 3D interchange format used by common 3D modeling and animation programs.

Introduction: What is VRML?

- VRML supports the creation of *virtual realities*. The VRML model:
 - Assumes that there is a real person (the viewer) who views, moves through, and interacts with the VRML world.
 - Provides the data needed by 3D rendering libraries to create a computer graphics facsimile of the VRML world.
 - Enables objects in the VRML world to be endowed with behaviors, and to have them linked together with one another and with the viewer so that all may interact in a realistic fashion.
 - Supports (but does not require) immersive virtual reality technologies and 3D input devices.

Introduction: VRML Files

- A VRML file is a textual description of a 3-dimensional virtual world. It can be created with any text editor, and its name should end with `.wrl`.
- When your graphical web browser reads a VRML file, it builds and presents the world described by the file. The browser also provides controls for moving through the world and for manipulating objects in the world.
- The browser interprets and runs the VRML file by loading a *VRML plug-in*. Examples include Silicon Graphics' *Cosmo Player* and Intervista's *WorldView*. For details visit the VRML Repository at:
<http://www.web3d.org/vrml/vrml.htm>

Key Concepts: Nodes

- A VRML file contains four main components:
 - A header, which describes the file to the browser.
 - Prototypes, which define new node types.
 - Nodes, which describe the shapes and other elements of the world.
 - Routes, which describe the behavior of the world.
- *Nodes* are the basic building blocks. Nodes are used to describe such things as shapes, colors, animation timers, animation interpolators, lights, and viewpoints. VRML 2.0 defines 54 node types, and includes a mechanism for building new (user-defined) node types.

Key Concepts: Nodes

- A node has these three characteristics:
 - *A type name*, such as Box, Sphere, Sound, or SpotLight.
 - *A set of fields* whose values distinguish a node from other nodes of the same type. For instance, the following Cone node:

```
Cone {  
    bottomRadius 1  
    height 8  
}
```

has fields named `bottomRadius` and `height` with values 1 and 8 respectively. This Cone node has other fields as well, assigned their default values.

- *A set of events* that the node can receive and send.

Key Concepts: Nodes

```
#VRML V2.0 utf8

# Try1.wrl (A simple VRML program)
# A cylinder of radius 0.5 and height 10.0. The cylinder is centered at
# the origin of world coordinate space, and extends (vertically)
# along the y axis.

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1.0 0.0 0.0 # red
    }
  }
  geometry Cylinder {
    radius 0.5
    height 10.0
  }
}
```

Key Concepts: Nodes

- In the previous slide:
 - The header indicates that this is a VRML 2.0 file encoded by the UCS Transform Format (a superset of ASCII):
`#VRML V2.0 utf8`
 - A Shape node has two fields: `appearance` and `shape`. We are providing values for each of these two fields.
 - The value of the `appearance` field is an Appearance node. In turn, the value of the Appearance node's `material` field is a Material node. In turn, the value of the Material node's `diffuseColor` field is the color red, represented as three fractions in the order red-green-blue. The Material node defines other fields as well, but these other fields' default values are used.
 - The value of the Shape's `geometry` field is a Cylinder node, whose `radius` and `height` fields are assigned values.

Key Concepts: Nodes

- The syntax for representing nodes in a file is as follows:

```
nodetype {  
    fields  
}
```

where `fields` is a list of zero or more pairs of this form:

```
fieldname fieldvalue
```

- The field names are specific to `nodetype` (for instance, `radius` is a field name for the `Sphere` node, but `length` is not).
- The order in which field names are given is irrelevant.
- Any fields not assigned values assume the default values supplied by the definition of `nodetype`.

Key Concepts: VRML Types

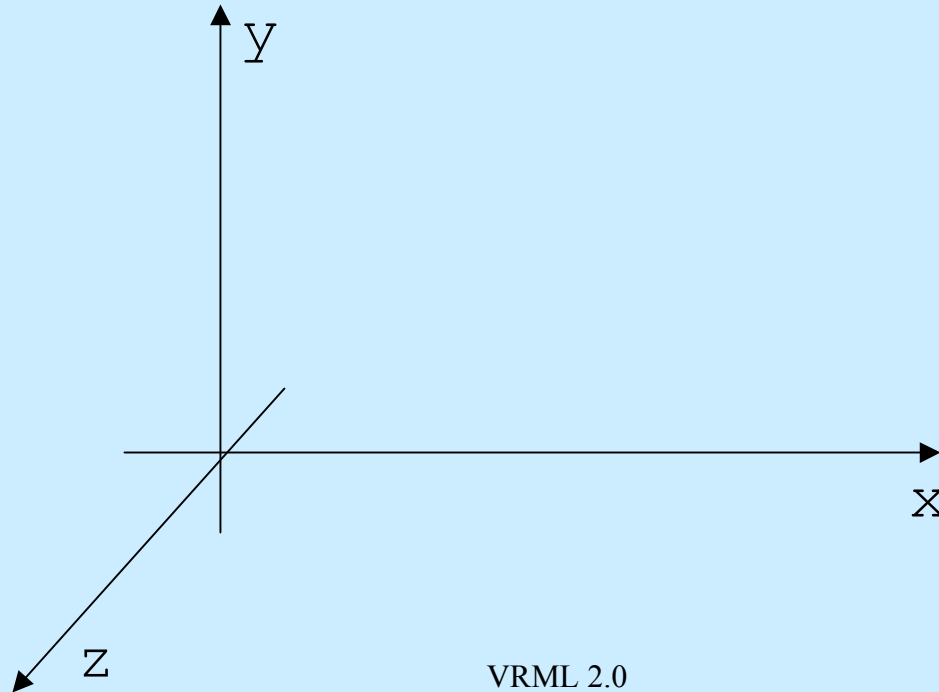
- *Fields* and *events* are typed. The type of a field determines the type of data it can hold. The type of an event is the type of data messages that it receives or sends.
- Types come in two flavors:
 - single-valued types—the names of such types begin with SF.
 - multiple-value types—the names of such types begin with MF.
- For example, `34` is a single-value integer type, whereas `[16 45 23]` is a multiple-value integer type.
- When specifying a multiple-value field type, the values are enclosed in square brackets. (If only one value is assigned to a multiple-value field, the square brackets may be omitted.)

Key Concepts: VRML Types

- SFBool A Boolean value.
- SFColor/MFColor An RGB color, or list of colors.
- SFFloat/MFFloat A floating-point, or list of floats.
- SFImage An image represented by integer values.
- SFInt32/MFInt32 A 32-bit integer, or list of integers.
- SFNode/MFNode A VRML node, or list of nodes.
- SFRotation/
MFRotation A rotation represented by four floats, or a list of such rotations
- SFString/MFString A string, or list of strings.
- SFTime A float time value (seconds since midnight 1/1/70).
- SFVec2f/MFVec2f A 2D vector of floats, or list of 2D vectors.
- SFVec3f/MFVec3f A 3D vector of floats, or list of 3D vectors.

Key Concepts: VRML Coordinate System

- VRML assumes a right-handed coordinate system in which the x axis is positive to the right, the y axis is positive upward, and the z axis is positive toward you in space. This is called the *world coordinate system* (WCS).



Key Concepts: VRML Wiring

- A VRML file may contain instructions for wiring nodes together, to give the world behavior. VRML wiring is specified through a set of `ROUTE` instructions. Each `ROUTE` instruction connects an `eventOut` of one node (the source node) to an `eventIn` of another node (the target node). The `eventOut` and `eventIn` must have the same type.
- Each `ROUTE` instruction specifies:
 - the pair of nodes to be wired together,
 - the output (called an *eventOut*) of the source node, and
 - the input (called an *eventIn*) of the target node.

Key Concepts: VRML Wiring

- The following wires the eventOut `fraction_changed` of a node named `Clock` to the eventIn named `set_fraction` of a node named `CubePath`:

```
ROUTE Clock.fraction_changed TO CubePath.set_fraction
```

- A node defines these kinds of fields and events :
 - An *eventIn* serves as an input jack (it receives incoming events).
 - An *eventOut* serves as an output jack (it sends outgoing events).
 - An *exposedField* holds a value, and serves as both an input jack and an output jack.
 - A *field* holds a value but does not serve as a jack.

Key Concepts: VRML Wiring

- An `exposedField` explicitly defines a field and implicitly defines an `eventIn` and an `eventOut` that connect to the field. For example, the declaration

```
exposedField foo
```

is equivalent to the declarations

```
field foo
```

```
eventIn set_foo
```

```
eventOut foo_changed
```

When a data value is sent to `set_foo`, the value is automatically assigned to the field `foo` and output through the `eventOut foo_changed`.

Key Concepts: VRML Wiring

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material DEF ThisColor Material {
      diffuseColor 1.0 0.0 0.0 # red
    }
  }
  geometry Cylinder { }
}
# other nodes defined here, including nodes named Clock and ColorPath.

# Here we wire Clock's eventOut fraction_changed to ColorPath's
# eventIn set_fraction.
ROUTE Clock.fraction_changed TO ColorPath.set_fraction

# And here we wire ColorPath's eventOut value_changed to
# ThisColor's implied eventIn set_diffuseColor.
ROUTE ColorPath.value_changed TO ThisColor.set_diffuseColor
```

Primitive Shapes: The Shape Node

- VRML primitive shapes are built from Shape nodes:

```
Shape {  
    appearance    NULL    # exposedField SFNode  
    geometry      NULL    # exposedField SFNode  
}
```

- A Shape node has two fields: appearance and geometry. Each field is an exposedField of type SFNode, and each has a default value of NULL.

Primitive Shapes: The Appearance Node

- Appearance nodes describe the appearance of shapes. If the appearance is described by material properties (color and shading) other than texture, then only the node's `material` field need be set.

```
Appearance {  
    material NULL          # exposedField SFNode  
    texture NULL          # exposedField SFNode  
    textureTransform NULL  
                          # exposedField SFNode  
}
```

Primitive Shapes: The Material Node

- A `Material` node is generally used as the value of the `Appearance` node's `material` field:

```
Material {
    ambientIntensity 0.2          # exposedField SFFloat
    diffuseColor 0.8 0.8 0.8     # exposedField SFColor
    emissiveColor 0.0 0.0 0.0    # exposedField SFColor
    shininess 0.2                # exposedField SFFloat
    specularColor 0.0 0.0 0.0    # exposedField SFColor
    transparency 0.0            # exposedField SFFloat
}
```

Primitive Shapes: The Material Node

- The fields of the `Material` node have this interpretation:
 - `ambientIntensity` specifies how much omnidirectional (i.e., atmospheric) light falls on the surface.
 - `diffuseColor` reflects all light sources depending on the angle between the light source and the surface. The more directly the surface faces the light source, the brighter the surface.
 - `specularColor` and `shininess` control the mirrorlike highlights, based on the orientation of the surface and the positions of the viewer and light sources.
 - `emissiveColor` models glowing, self-luminous surfaces.
 - `transparency` specifies the percentage of light that is transmitted by the surface, where 0.0 is completely opaque and 1.0 is completely transparent.

Primitive Shapes: Building Shapes

- The primitive *geometries* are provided by `Box`, `Cone`, `Cylinder`, and `Sphere` nodes. These are all *solid geometries*. (In addition to these nodes, VRML provides several node types for building new geometries.)
- Each primitive geometry node has fields specific to the geometry, such as a `radius` field for setting the radius of a sphere, and a `size` field for setting the dimensions of a box.
- Primitive shapes are always centered at the origin. Later, we will look at `Transform` nodes, which are used to move the origin, thereby enabling us to build primitive shapes at any location.

Primitive Shapes: The BOX Node

- A BOX node builds a box centered at the origin and aligned with the axes. The default value of its (sole) field `size` is the *SFVec3f* $(2, 2, 2)$, meaning that the default box has length 2 units per side:

```
Box {  
    size 2.0 2.0 2.0 # field SFVec3f  
}
```

Primitive Shapes: The Cone Node

- A cone, specified by its bottom radius and its height, is centered at the origin and aligned along the vertical *y* axis. Fields *side* and *bottom* are provided to specify whether the cone is to possess its side faces and its bottom face respectively:

```
Cone {  
    bottomRadius    1.0    # field SFFloat  
    height          2.0    # field SFFloat  
    side            TRUE   # field SFBool  
    bottom          TRUE   # field SFBool  
}
```

Primitive Shapes: The Cylinder Node

- A cylinder, specified by its radius and height, is centered at the origin and aligned along the *y* axis. Fields *side*, *top*, and *bottom* are provided to specify whether the cylinder is to possess its side faces, top face, and bottom face respectively:

```
Cylinder {  
    radius          1.0    # field SFFloat  
    height         2.0    # field SFFloat  
    side           TRUE   # field SFBool  
    top            TRUE   # field SFBool  
    bottom         TRUE   # field SFBool  
}
```

Primitive Shapes: The Sphere Node

- A sphere is centered at the origin and of specified radius:

```
Sphere {  
    radius          1.0    # field SFFloat  
}
```

Primitive Shapes: Examples

- A blue cone whose axis is aligned with the y axis, centered at the origin, of height 2 and bottom radius 3. Its bottom face lies on the plane $y=-1$, and the apex at the point $(0, 1, 0)$.

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material
      diffuseColor 0.0 0.0 1.0 # blue
  }
}
geometry Cone {
  bottomRadius 3.0
}
}
```

Primitive Shapes: Examples

- A white cylinder whose axis is aligned with the y axis, centered at the origin, of radius 4 and height 10.5:

```
#VRML 2.0 utf8
Shape {
  appearance Appearance {
    material Material { }
  }
  geometry Cylinder {
    radius 4.0
    height 10.5
  }
}
```

Primitive Shapes: Examples

- A green box, centered at the origin, of length 1 along the x axis, 2 along the y axis, and 4 along the z axis.

```
#VRML 2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.0 1.0 0.0
    }
  }
  geometry Box {
    size 1.0 2.0 4.0
  }
}
```

Primitive Shapes: Examples

- A cyan sphere, centered at the origin and of radius 1.

```
#VRML 2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0 1 1
    }
  }
  geometry Sphere { }
```

Primitive Shapes: The Group Node

- The Group node is used to group together any number of shapes. The shapes, enclosed by square brackets [], are the value of the Group node's children field:

```
Group {
  children      []      # exposedField MFNode
  bboxCenter    0.0 0.0 0.0
                # field SFVec3f
  bboxSize      -1.0 -1.0 -1.0
                # field SFVec3f
  addChildren   # eventIn MFNode
  removeChildren # eventIn MFNode
}
```

Primitive Shapes: Examples

- A cross of magenta boxes, along the x and y axes:

```
#VRML V2.0 utf8
Group {
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1 0 1
        }
      }
      geometry Box { size 20 2 2 }
    }, Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1 0 1
        }
      }
      geometry Box { size 2 20 2 }
    }
  ]
}
```

Primitive Shapes: Instancing Nodes

- A node can be *instanced* any number of times. To instance a node, it must first be named. Once named, the node can be referenced any number of times in the remainder of the same file.
- To name a node, type the keyword `DEF`, followed by a name, followed by the node to be named.
- To reference a named node, type the keyword `USE`, followed by the name of the node being referenced.
- You can reference a node only in a context that allows the type of node being referenced.
- Nodes used in `ROUTE` statements must be named.

Primitive Shapes: Instancing Nodes

- A cross of magenta boxes, along the x and y axes. Using the DEF syntax, name an Appearance node; using the USE syntax, we reference the named node:

```
#VRML V2.0 utf8
Group {
  children [
    Shape {
      appearance DEF Magenta Appearance {
        material Material {
          diffuseColor 1 0 1
        }
      }
      geometry Box { size 20 2 2 }
    }, Shape {
      appearance USE Magenta
      geometry Box { size 2 20 2 }
    }
  ]
}
```

Transforming Shapes: Transform Nodes

- In VRML, you can create any number of coordinate systems. Each coordinate system is positioned, oriented, and scaled relative to another coordinate system. The new coordinate system is a *child coordinate system* that is nested within its *parent's coordinate system*. The parent in turn can be a child of yet another coordinate system.
- The coordinate system at the root of the hierarchy is the *world coordinate system (WCS)*. Every VRML world is ultimately defined in the world coordinate system.
- New coordinate systems are defined using the `Transform` node.

Transforming Shapes: Transform Nodes

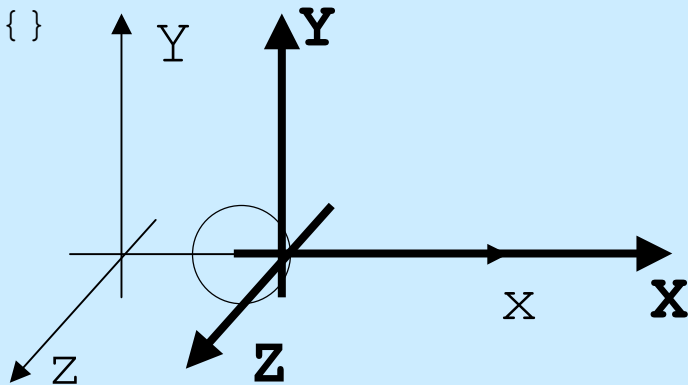
```
Transform {
  children      []          # exposedField MFNode
  translation   0.0 0.0 0.0 # exposedField SFVec3f
  rotation      0.0 0.0 0.0 0.0 # exposedField SFRotation
  scale         1.0 1.0 1.0   # exposedField SFVec3f
  scaleOrientation 0.0 0.0 1.0 0.0
                                     # exposedField SFRotation
  bboxCenter    0.0 0.0 0.0   # field SFVec3f
  bboxSize      -1.0 -1.0 -1.0 # field SFVec3f
  center        0.0 0.0 0.0   # exposedField SFVec3f
  addChildren   # eventIn MFNode
  removeChildren # eventIn MFNode
}
```

Transforming Shapes: Translation

- You can use a `Transform` node to build a new coordinate system that is *translated* with respect to an existing coordinate system. When a primitive shape is built in the new coordinate system, it is centered at the new coordinate system's origin. The VRML program presented in the following slide constructs a sphere whose center is at $(2, 0, 0)$ in world coordinates.
- A `Transform` node is more general than a `Group` node: A `Transform` node groups its children in a new coordinate system, whereas a `Group` node groups its children in the old coordinate system, the one in which the `Group` node itself is defined.

Transforming Shapes: Translation

```
#VRML V2.0 utf8
# translate by 2 units along the x axis
Transform {
  translation 2 0 0
  children Shape {
    appearance Appearance {
      material Material {}
    }
    geometry Sphere {}
  }
}
```

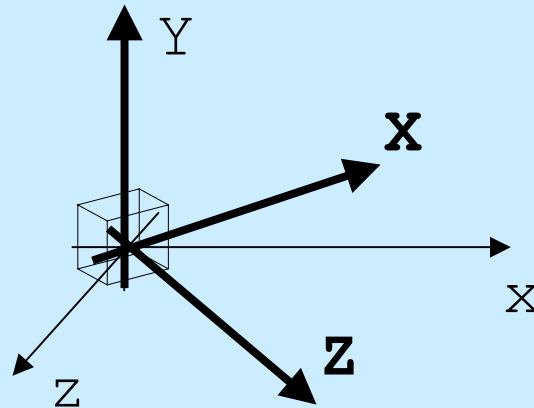


Transforming Shapes: Rotation

- You can use a `Transform` node to build a new coordinate system that is rotated with respect to an existing coordinate system. When a primitive shape is built in the new coordinate system, the shape is centered at the old coordinate system's origin, but rotated.
- A rotation is specified by an *SFRotation* value, a list of four floating-point values. The first three values represent the vector about which rotation is to occur; the fourth value represents the angle of rotation in radians. Rotation follows the right-handed rule: when viewed down the vector of rotation, the rotation appears counter-clockwise.

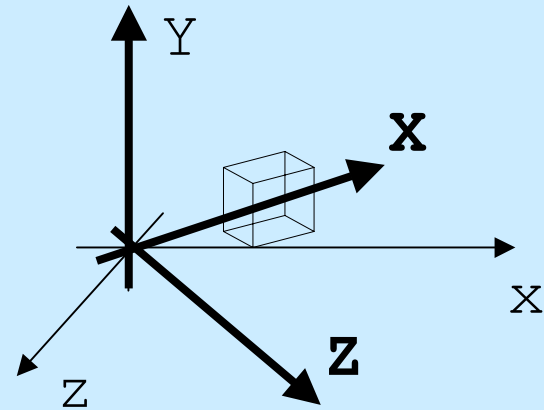
Transforming Shapes: Rotation

```
#VRML V2.0 utf8
# rotate by 45 degrees about the y axis
Transform {
  rotation 0.0 1.0 0.0 0.785 # .785 radians = 45 degrees
  children Shape {
    appearance Appearance {
      material Material { }
    }
    geometry Box { }
  }
}
```



Transforming Shapes: Rotation

```
#VRML V2.0 utf8
# rotate by 45 degrees about the y axis, then translate
# along the x axis in the rotated coordinate system.
Transform {
  rotation 0.0 1.0 0.0 0.785 # 45 degrees
  children Transform {
    translation 6.0 0.0 0.0
    children Shape {
      appearance Appearance {
        material Material { }
      }
      geometry Box { }
    }
  }
}
```

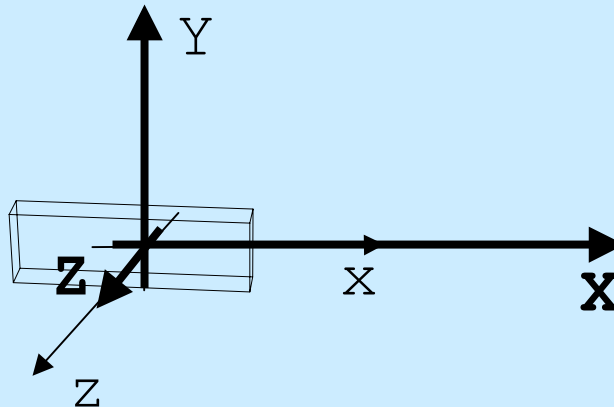


Transforming Shapes: Scale

- You can use a `Transform` node to build a new coordinate system that is scaled with respect to an existing coordinate system. When a primitive shape is built in the new coordinate system, the shape is centered at the old coordinate system's origin, but scaled.
- A scale is specified by an *SFVec3f* value, a list of three floating-point values that specify the scale factor along the x, y, and z axes respectively. The scale is *uniform* if all three values are equal, and *non-uniform* otherwise.

Transforming Shapes: Scale

```
#VRML V2.0 utf8
# scale by 2, 1, and 0.5 along the x, y, and z axes
Transform {
  scale 2.0 1.0 0.5
  children Shape {
    appearance Appearance {
      material Material { }
    }
    geometry Box { }
  }
}
```

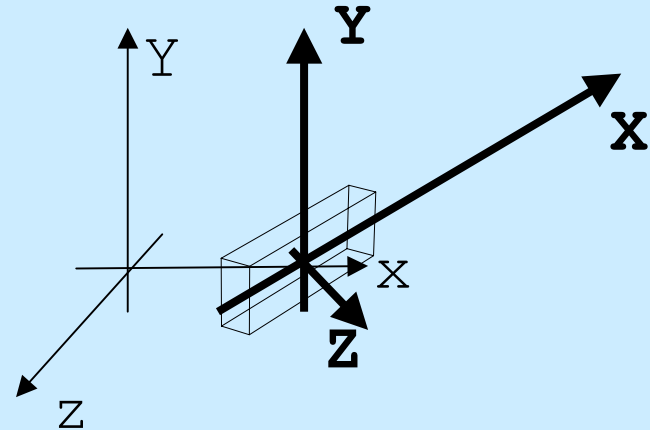


Transforming Shapes: Combining Transforms

- The value of the `Transform` node's `center` field is the point about which rotations and scales are to occur. By default, this value is the origin $(0, 0, 0)$.
- A `Transform` node combines the various transformations as follows:
 - scales about the center point, then
 - rotates about the center point, then
 - translates relative to the parent's coordinate system (that is, relative to the coordinate system prior to scale and rotation).
 - (Note: The `scaleOrientation` field is used to temporarily rotate the object's coordinate system in preparation for a scale, then rotates back. It is useful for scaling along a direction not aligned with the local coordinate axes, and is rarely used.)

Transforming Shapes: Combining Transforms

```
#VRML V2.0 utf8
# scale, rotate, and translate
Transform {
  scale 2.0 1.0 0.5
  rotation 0.0 1.0 0.0 0.785 # 45 degrees
  translation 4.0 0.0 0.0
  children Shape {
    appearance Appearance {
      material Material { }
    }
    geometry Box { }
  }
}
```



Animating Worlds

- Many features in a VRML world can be animated. Here we focus on animating the position, orientation, and scale of shapes.
- To start and stop an animation, we use a `TimeSensor` node, which behaves like a clock.
- The time events generated by a `TimeSensor` node are routed into an *interpolator node*, which outputs new change-in-position or change-in-orientation events. VRML provides several kinds of interpolator nodes.
- The events produced by an interpolator node are routed into a `Transform` node to reposition, reorient, or rescale the `Transform` node's coordinate system.

Animating Worlds : TimeSensor Nodes

- The `TimeSensor` node, a generator of new events, serves as a clock. While turned on, it outputs events through its `fraction_changed` and `time` eventOuts. These events can be routed into other nodes (typically interpolator nodes) to achieve certain effects.
- *Absolute time*, output by the `time` eventOut, is measured in seconds since midnight of 1/1/70 (GMT).
- *Fractional time*, output by `fraction_changed`, is a fraction in the range `[0 . . 1]` indicating what portion of the `cycleInterval` has transpired. For instance, if `cycleInterval` is 8 seconds and 2 seconds have passed, the value 0.25 is sent by `fraction_changed`.

Animating Worlds: TimeSensor Nodes

```
TimeSensor {
  enabled          TRUE      # exposedField SFBool
  startTime        0.0       # exposedField SFTIME
  stopTime         0.0       # exposedField SFTIME
  cycleInterval    1.0       # exposedField SFTIME
  loop             FALSE     # exposedField SFBool
  isActive         # eventOut SFBool
  time             # eventOut SFTIME
  cycleTime        # eventOut SFTIME
  fraction_changed # eventOut SFFloat
}
```

Animating Worlds :

PositionInterpolator Nodes

- VRML defines several kinds of interpolator nodes. Each inputs a fraction f into its `set_fraction` eventIn, and outputs a value through its `value_changed` eventOut. The output value is derived from f by interpolation. Interpolator nodes often receive fractional time from a clock, and map this time to a useful value which it outputs to yet another node.
- A `PositionInterpolator` node inputs a fraction, and outputs a value of type *SFVec3f*. This value is typically used as a position or a scale value, to animate position or scale.

Animating Worlds :

PositionInterpolator Nodes

```
PositionInterpolator {  
  key          []      # exposedField MFFloat  
  keyValue     []      # exposedField MFVec3f  
  set_fraction # eventIn   SFFloat  
  value_changed # eventOut  SFVec3f  
}
```

- When a `PositionInterpolator` node receives a fractional time through `set_fraction`, it computes a position based on its list of keys (fractions) and key values (positions). Given fractional time t , the node:
 - scans the list `key` for two adjacent times t_1 and t_2 such that $t_1 \leq t \leq t_2$, then
 - retrieves the corresponding pair of positions p_1 and p_2 from the list `keyValue`, then
 - computes an intermediate position by linearly interpolating between p_1 and p_2 , which it outputs via `value_changed`.

Animating Worlds :

OrientationInterpolator Nodes

- An `OrientationInterpolator` node inputs a fraction, and outputs a value of type *SFRotation*. This output value is typically used as a rotation value, to animate a shape's orientation. (Recall that an *SFRotation* value is a list of four floating-point values: the first three determine the axis of rotation, and the fourth determines the angle of rotation in radians. For example, the *SFRotation* value

1.0 0.0 0.0 1.571

corresponds to a rotation of $\pi/2$ radians (90 degrees) around the x axis.)

Animating Worlds :

OrientationInterpolator Nodes

```
OrientationInterpolator {  
  key          []      # exposedField MFFloat  
  keyValue     []      # exposedField MFRotation  
  set_fraction # eventIn   SFFloat  
  value_changed # eventOut  SFRotation  
}
```

- When a `OrientationInterpolator` node receives a fractional time through `set_fraction`, it computes a position based on its list of keys (fractions) and key values (orientations). Given fractional time t , the node:
 - scans the list `key` for two adjacent times t_1 and t_2 such that $t_1 \leq t \leq t_2$, then
 - retrieves the corresponding pair of `SFRotations` r_1 and r_2 from the list `keyValue`, then
 - computes an intermediate `SFRotation` by linearly interpolating between r_1 and r_2 , which it outputs via `value_changed`.

Animating Worlds: Examples

- The following world (over this slide and the next) moves a unit cube along a square path that lies in the horizontal xz plane.

```
#VRML V2.0 utf8
# the cube
DEF Cube Transform {
  children Shape {
    geometry Box { size 1 1 1 }
    appearance Appearance {
      material Material {}
    }
  }
},
```

Animating Worlds: Examples (continued from previous slide)

```
# animation clock
DEF Clock TimeSensor {
    cycleInterval 8.0      # 8 seconds
    loop TRUE             # loop forever
},
# animation path
DEF CubePath PositionInterpolator {
    key [ 0.0  0.25  0.5  0.75  1.0 ]
    keyValue [
        2 0  2,          2 0 -2,
        -2 0 -2,         -2 0  2,          2 0  2
    ]
}
ROUTE Clock.fraction_changed TO CubePath.set_fraction
ROUTE CubePath.value_changed TO Cube.set_translation
```

Animating Worlds: Examples

- The following world (over this slide and the next) spins a unit cube around its diagonal that is aligned in the direction $(1, 1, 1)$.

```
#VRML V2.0 utf8
# the cube
DEF Cube Transform {
  children Shape {
    geometry Box { size 1 1 1 }
    appearance Appearance {
      material Material {}
    }
  }
},
```

Animating Worlds: Examples (continued from previous slide)

```
# animation clock
DEF Clock TimeSensor {
    cycleInterval 4.0      # 4 seconds
    loop TRUE            # loop forever
},
# animation orientation
DEF CubeOrientation OrientationInterpolator {
    key [ 0.0  0.5  1.0]
    keyValue [
        1 1 1 0.0,      1 1 1 3.14,      1 1 1 6.28
    ]
}
ROUTE Clock.fraction_changed TO CubeOrientation.set_fraction
ROUTE CubeOrientation.value_changed TO Cube.set_rotation
```

Sensing Viewer Actions: Sensors

- To make your world interactive, you can attach to your shapes sensors that detect viewer actions, such as clicking or dragging with the mouse. There are four sensor nodes:
 - The *touch sensor* node `TouchSensor` detects when the viewer moves the mouse over or clicks the shape, and outputs when and where the touch occurred.
 - The *motion sensor* nodes `CylinderSensor`, `PlaneSensor`, and `SphereSensor` detect when the viewer clicks and drags the shape. These motion sensors output positions based on a particular geometry. For instance, while dragging a shape to which a `PlaneSensor` is attached, the `PlaneSensor` outputs positions across a plane parallel to the local `xy` plane.
- We'll cover the `TouchSensor`, `PlaneSensor`, and `SphereSensor` nodes.

Sensing Viewer Actions: Sensors

- A sensor node is added to any group, such as those created by the `Group` and `Transform` nodes. When in such a group, the sensor detects user actions applied to any shape in the same group. Such shapes are collectively called the *sensor's geometry*. When a sensor detects a user action, it generates events through its `eventOuts`.
- Sensors are named by `DEF` and used in `ROUTE` statements. The `eventOuts` of `TouchSensors` are often routed to initiate behaviors, much as `TimeSensors` are used. The `eventOuts` of motion sensors are typically routed to named `Transform` nodes to change the values of `rotation`, `translation`, or `scale` fields.

Sensing Viewer Actions: TouchSensor Nodes

```
TouchSensor {  
    enabled          TRUE      # exposedField SFBool  
    isActive         # eventOut SFBool  
    isOver           # eventOut SFBool  
    touchTime        # eventOut SFTime  
    hitPoint_changed # eventOut SFVec3f  
    hitNormal_changed # eventOut SFVec3f  
    hitTexCoord_changed # eventOut SFVec2f  
}
```

- When the cursor moves over a shape in the TouchSensor's group, TRUE is sent over the isOver eventOut; when it moves off the shapes in the group, FALSE is sent over the isOver eventOut.
- When the user *drags* over a shape in the TouchSensor's group, TRUE is sent over isActive; when the user releases, FALSE is sent over isActive.

Sensing Viewer Actions: Examples

- The following world (over this slide and the next) rotates a cube around its *y* axis whenever the cursor is over it (the mouse button need not be pressed).

```
#VRML V2.0 utf8
DEF Cube Transform {
  children [
    Shape {
      appearance Appearance {
        material Material { }
      }
      geometry Box { }
    },
    DEF Touch TouchSensor { }
  ]
},
```

Sensing Viewer Actions: Examples (continued from previous slide)

```
DEF Clock TimeSensor {
  enabled FALSE
  cycleInterval 4.0
  loop TRUE
},
DEF CubePath OrientationInterpolator {
  key [ 0.0, 0.50, 1.0 ]
  keyValue [
    0.0 1.0 0.0 0.0,
    0.0 1.0 0.0 3.14,
    0.0 1.0 0.0 6.28 ]
}
ROUTE Touch.isOver TO Clock.set_enabled
ROUTE Clock.fraction_changed TO CubePath.set_fraction
ROUTE CubePath.value_changed TO Cube.set_rotation
```

Sensing Viewer Actions: Examples

- The following world (over this slide and the next) rotates a cube once around its *y* axis whenever the user presses while the cursor is over it.

```
#VRML V2.0 utf8
```

```
Group {  
  children [  
    DEF Cube Transform {  
      children Shape {  
        appearance Appearance {  
          material Material { }  
        }  
        geometry Box { }  
      }  
    },  
    DEF Touch TouchSensor { },
```

Sensing Viewer Actions: Examples (continued from previous slide)

```
DEF Clock TimeSensor { cycleInterval 4.0 },
DEF CubePath OrientationInterpolator {
    key [ 0.0, 0.50, 1.0 ]
    keyValue [
        0.0 1.0 0.0 0.0,
        0.0 1.0 0.0 3.14,
        0.0 1.0 0.0 6.28
    ]
}

]
}

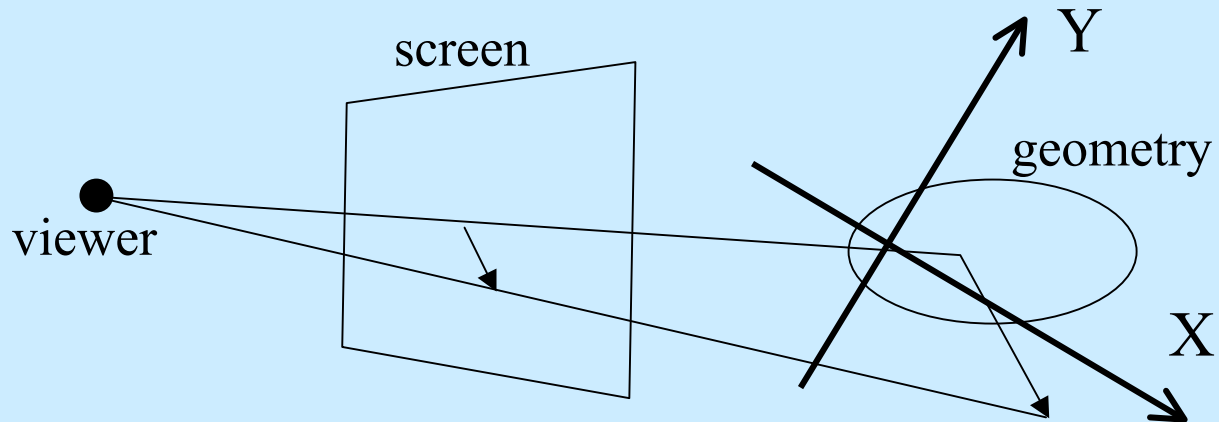
ROUTE Touch.touchTime TO Clock.set_startTime
ROUTE Clock.fraction_changed TO CubePath.set_fraction
ROUTE CubePath.value_changed TO Cube.set_rotation
```

Sensing Viewer Actions: PlaneSensor Nodes

- `PlaneSensor` nodes are used to detect viewer actions and convert them to outputs suitable for translating shapes along a plane.
- In typical use, the children of a `Transform` node include a `PlaneSensor` node and one or more shapes (the sensor's geometry). When the geometry is dragged, it follows the cursor across a plane parallel to the xy plane of its local coordinate system. To achieve this, a *SFVec3F* sent through the `PlaneSensor` node's `translation_changed` `eventOut` is routed to the `Transform` node's `set_translation` `impliedEventIn`.

Sensing Viewer Actions: PlaneSensor Nodes

- When the mouse button clicks down over the sensor's geometry, an initial *intersection point* is calculated, on the surface of the geometry. Subsequent dragging motion is mapped to the plane that contains this intersection point and is parallel to the geometry's local x y plane.



Sensing Viewer Actions: PlaneSensor Nodes

```
PlaneSensor {
    enabled          TRUE          # exposedField SFBool
    autoOffset       TRUE          # exposedField SFBool
    offset           0.0 0.0 0.0   # exposedField SFVec3f
    maxPosition      -1.0 -1.0     # exposedField SFVec2f
    minPosition      0.0 0.0       # exposedField SFVec2f
    isActive         # eventOut SFBool
    translation_changed # eventOut SFVec3f
    trackPoint_changed # eventOut SFVec3f
}
```

- When the user drags over a shape in the PlaneSensor's group, TRUE is sent over isActive; when the user releases, FALSE is sent over isActive.
- While the viewer drags, SFVec3f's are sent through the translation_changed eventOut, reflecting where the cursor is over the plane parallel to the local xy plane.

Sensing Viewer Actions: Examples

- While the viewer drags the box, it slides across the xy plane, following the cursor.

```
#VRML V2.0 utf8
Group {
  children [
    DEF Cube Transform {
      children Shape {
        appearance Appearance {
          material Material { }
        }
        geometry Box { }
      }
    },
    DEF Sensor PlaneSensor { }
  ]
}
ROUTE Sensor.translation_changed TO Cube.set_translation
```

Sensing Viewer Actions: SphereSensor Nodes

- SphereSensor nodes are used to detect viewer actions and convert them to outputs suitable for rotating shapes around the center of their parent group's origin.
- In typical use, the children of a Transform node include a SphereSensor node and one or more shapes. When one of the shapes in the sensor's geometry is dragged, the shape rotates. To achieve this, the SFVec3Fs sent through the SphereSensor node's rotation_changed eventOut are routed to the Transform node's set_rotation implied eventIn.

Sensing Viewer Actions: SphereSensor Nodes

```
SphereSensor {  
    enabled          TRUE          # exposedField SFBool  
    autoOffset      TRUE          # exposedField SFBool  
    offset          0.0 1.0 0.0 0.0  
                                # exposedField SFRotation  
    isActive        # eventOut SFBool  
    rotation_changed # eventOut SFRotation  
    trackPoint_changed # eventOut SFVec3f  
}
```

- When the user drags over a shape in the SphereSensor's group, TRUE is sent over isActive; when the user releases, FALSE is sent over isActive.
- While the viewer drags, SFRotations are sent through the rotation_changed eventOut, reflecting the cursor's position over an imaginary sphere centered at the local coordinate system's origin.

Sensing Viewer Actions: Examples

- While the viewer drags either shape, both get rotated around the origin of the parent group's coordinate system (which is the world coordinate system).

```
#VRML V2.0 utf8
```

```
DEF Shape1 Transform {  
  children [  
    Shape {  
      appearance DEF White Appearance {  
        material Material { }  
      }  
      geometry Box { }  
    },
```

Sensing Viewer Actions: Examples (continued from previous slide)

```
DEF Shape2 Transform {
    translation 3 0 0
    children Shape {
        appearance USE White
        geometry Cone { }
    }
},
DEF Sensor SphereSensor { }
]
}
```



```
ROUTE Sensor.rotation_changed TO Shape1.set_rotation
ROUTE Sensor.rotation_changed TO Shape2.set_rotation
```

Sample VRML Programs

- In the next section I provide a number of sample VRML programs that illustrate the ideas and node types covered by the previous sections. These programs work, and are worthy of studying and of entering and running in a browser.

Sample VRML Programs

```
#VRML V2.0 utf8
# ThreeBoxes.wrl
# A group of three red boxes
Group {
  children [
    Shape {
      appearance DEF Red Appearance {
        material Material {
          diffuseColor 1 0 0 # red
        }
      }
      geometry Box {
        size 18 1 1
      }
    },
    Shape {
      appearance USE Red
      geometry Box {
        size 1 18 1
      }
    },
    Shape {
      appearance USE Red
      geometry Box {
        size 1 1 18
      }
    }
  ]
}
```

Sample VRML Programs

```
#VRML V2.0 utf8
# Fun1.wrl (continued on next slide)
# The red box's coordinate system is rotated 90 degrees
# around the y axis, then translated by (4,0,0) relative to
# its parent's coordinate system (the WCS). It ends up centered at (4,0,0).
# The blue box's coordinate system is rotated 90 degrees around the y axis,
# then translated by (4,0,0) in its local coordinate system.
# Since the blue box's local x axis coincides with the
# world coordinate system's (WCS) -z axis (after the rotation),
# the blue box ends up centered at (0,0,-4) in WCS.
```

```
Group {
  children [
    # the red box
    Transform {
      translation 4 0 0
      rotation 0 1 0 1.571
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor 1 0 0
          }
        }
        geometry DEF WideBox Box {
          size 2 0.5 0.5
        }
      }
    }
  ]
}
```

Sample VRML Programs

```
# Fun1.wrl (continued from previous slide)

# the blue box
Transform {
  rotation 0 1 0 1.571
  children Transform {
    translation 4 0 0
    children Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 0 1
        }
      }
      geometry USE WideBox
    }
  }
}
]
}
```

Sample VRML Programs

```
#VRML V2.0 utf8
# Pillars.wrl (continued on next slide)
Group {
  children [
    # the base
    Shape {
      appearance Appearance {
        material Material {}
      }
      geometry Box {
        size 11 1 11
      }
    }
    # front row of pillars
    Transform {
      translation 0 2.5 4
      children DEF Row Group {
        children [
          DEF Pillar Shape {
            appearance Appearance {
              material Material {
                diffuseColor 1 .3 .5
              }
            }
            geometry Cylinder {
              radius 0.4
              height 5.0
            }
          }
        ]
      }
    }
  ]
}
```

Sample VRML Programs

```
# Pillars.wrl (continued from previous slide)
    Transform {
        translation -2 0 0
        children USE Pillar
    }
    Transform {
        translation -4 0 0
        children USE Pillar
    }
    Transform {
        translation 2 0 0
        children USE Pillar
    } ] } }
# back row of pillars
Transform {
    translation 2 2.5 -4
    children USE Row
}
# right row of pillars
Transform {
    translation 4 2.5 0
    rotation 0 1 0 1.571
    children USE Row
}
# left row of pillars
Transform {
    translation -4 2.5 0
    rotation 0 1 0 -1.571
    children USE Row
}
] }
```

Sample VRML Programs

```
#VRML V2.0 utf8
# Spiral.wrl (continued on next slide)
# A spiral of spheres
Group {
  children [
    # 90 degree arc
    DEF Arc Group {
      children [
        Transform {
          translation 4 0 0
          children Shape {
            appearance Appearance {
              material Material {
                diffuseColor 1 1 0 # yellow
              }
            }
            geometry Sphere {}
          }
        }
      ]
    }
    Transform {
      translation 4 1 0
      center -4 0 0
      rotation 0 1 0 0.524
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor 1 0 1 # magenta
          }
        }
        geometry Sphere {}
      }
    }
  ]
}
```

Sample VRML Programs

```
# Spiral.wrl (continued from previous slide)
  Transform {
    translation 4 2 0
    center -4 0 0
    rotation 0 1 0 1.048
    children Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 1 1 # cyan
        }
      }
      geometry Sphere {}
    } ] }
# additional arcs
Transform {
  translation 0 3 0
  rotation 0 1 0 1.571
  children USE Arc
}
Transform {
  translation 0 6 0
  rotation 0 1 0 3.142
  children USE Arc
}
Transform {
  translation 0 9 0
  rotation 0 1 0 4.712
  children USE Arc
}
]
}
```

Sample VRML Programs

```
#VRML V2.0 utf8
# LongSpiral.wrl
# A spiral over three revolutions (inlines file Spiral.wrl
# of previous slide)
Group {
  children [
    DEF Spiral Inline {
      url "Spiral.wrl"
    }
    Transform {
      translation 0 12 0
      children USE Spiral
    }
    Transform {
      translation 0 24 0
      children USE Spiral
    }
  ]
}
```

References

- *The Annotated VRML 2.0 Reference Manual*, by R. Carey and G. Bell, Addison-Wesley, 1997. This excellent book can be found online at:
<http://www.wasabisoft.com/Book/book.shtml>
- *Moving Worlds*, by E. Adams and D. Doherty, Prima Publishing, 1996.
- *VRML 2.0 Sourcebook*, by A. Ames, D. Nadeau, and Java. Moreland, John Wiley & Sons, 1997.
- *The VRML Handbook*, by J. Hartman and J. Wernecke, Addison-Wesley, 1996.