

Introduction to VRML 2.0

Computer Graphics part 2

Michael Lund, ph.d. student



Lecture overview

Introduction to scenegraph-based 3D graphics programming (*VRML 2.0*)

Topics will include:

- What VRML 2.0 is
- VRML file structure
- Concepts and terminology
- Some syntax
- Where to find out more



Course material

- The Annotated VRML97 Reference Manual (R. Carey & G. Bell 1997). Handout.
Chapter 1, 2.1-2.4, 2.6-2.11
<http://doc.sch130.nsc.ru/www.wasabisoft.com/Book/Book.html>
- SIGGRAPH 1996 VRML course material. Optional.
International Conference on Computer Graphics and Interactive Techniques



What is VRML?

- VRML is:
 - An acronym for Virtual Reality Modeling Language



VRML is neither *virtual reality* nor a *modeling language*

- VR typically implies an immersive 3D experience (e.g. head-mounted display)
- VRML neither requires nor precludes immersion.
- VRML provides a bare minimum of geometric modeling features
- VRML provides features beyond the scope of a modeling language



? ? ? ?

- VRML is:
 - 3D interchange format
 - » A simple text file language for describing 3-D shapes and interactive environments
 - A web standard
 - » A 3D analog to HTML
 - A technology that integrates 3D, 2D, text, audio, and multimedia into a coherent model.



Writing and Viewing VRML Files

- You can write VRML files using:
 - Any text editor
 - Many *world builder* applications
- You can view VRML files using:
 - A VRML browser
 - A VRML plug-in to an HTML browser
- VRML is endorsed by:
 - Most 3-D graphics vendors
 - Most web browser vendors



Major features of VRML

- Scene graph structure
- Event Architecture
- Sensors
- Script and interpolators
- Prototyping
- Distributed Scenes



Scene graph structure

- 3D object and worlds are described using a hierarchical scene graph . The scenegraph is a directed acyclic graph.
- Entities in the scene are called *nodes*. VRML 2.0 defines 54 different node types, including geometry primitives, appearance properties, sound and sound properties, and various types of grouping nodes.
- Nodes store their data in *fields*. VRML 2.0 have 20 different types of fields that can be used.
- Scenegraph structure makes it easy to create complicated objects from subparts.



Event Architecture

- VRML 2.0 defines an *event* mechanism by which nodes in the scene graph can communicate with each other.
- Each node type defines the names and types of events that instances of that type may generate or receive
- ROUTE statements define paths between event generators and receivers



Sensors

- *Sensors* are the basic interaction and animation primitives of VRML.
- The **TimeSensor** node generates events as time passes and is basic for all animated behaviors.
- Other sensors are the basis for user interaction, generating events as the user interacts with the input devices or moves through the world.
- Sensors only generate events, they must be combined with other nodes via ROUTE statements to have any visible effect on the scene.



Scripts

- Script nodes can be inserted between event generators and receivers.
- Scripts allow the world creator to define arbitrary behaviors, defined in any supporting scripting language
- VRML 2.0 defines Script node bindings for Java and JavaScript



Prototyping

- The PROTO statement is a prototyping mechanism for encapsulating and reusing a scene graph.
- Geometry, properties, and animations can be encapsulated, either separately or together.



Distributed Scenes

- VRML 2.0 allow a single VRML world definition to span the WWW
- The inline node allows the inclusion of another VRML file stored anywhere on the Web
-
- EXTERNPROTO statement allows new node definitions to be fetched from anywhere on the Web



VRML File Structure

- VRML files contain:
 - The file header
 - *Comments* - notes to yourself
 - *Nodes* - nuggets of scene information
 - *Fields* - node attributes you can change
 - *Values* - attribute values
 - *Node Names* - names for reusable nodes
 - more. . .



The file header

- Every VRML file shall begin with:
 - `#VRML V2.0 utf8`
- The text identifies the file as a VRML file and the encoding of the file
- The identifier `utf8` indicates a clear text encoding that allows for international characters to be displayed (known as Unicode).
- In the UTF-8 encoding, `#` begins a comment. Only the first comment (the header) has semantic meaning.



Primitive Shapes

- Shapes are the building blocks of a VRML world
- Primitive Shapes are the standard building blocks:
 - Box
 - Cone
 - Cylinder
 - Sphere



Shape nodes

- Shape nodes describe:
 - *geometry* - form, or structure
 - *appearance* - color and texture
- Shape {
 - geometry . . .
 - appearance . . .}



Geometry nodes

- Shape geometry is built with *geometry* nodes
- Standard, *primitive* geometry nodes include:

Box { . . . }

Cone { . . . }

Cylinder { . . . }

Sphere { . . . }



Geometry node fields

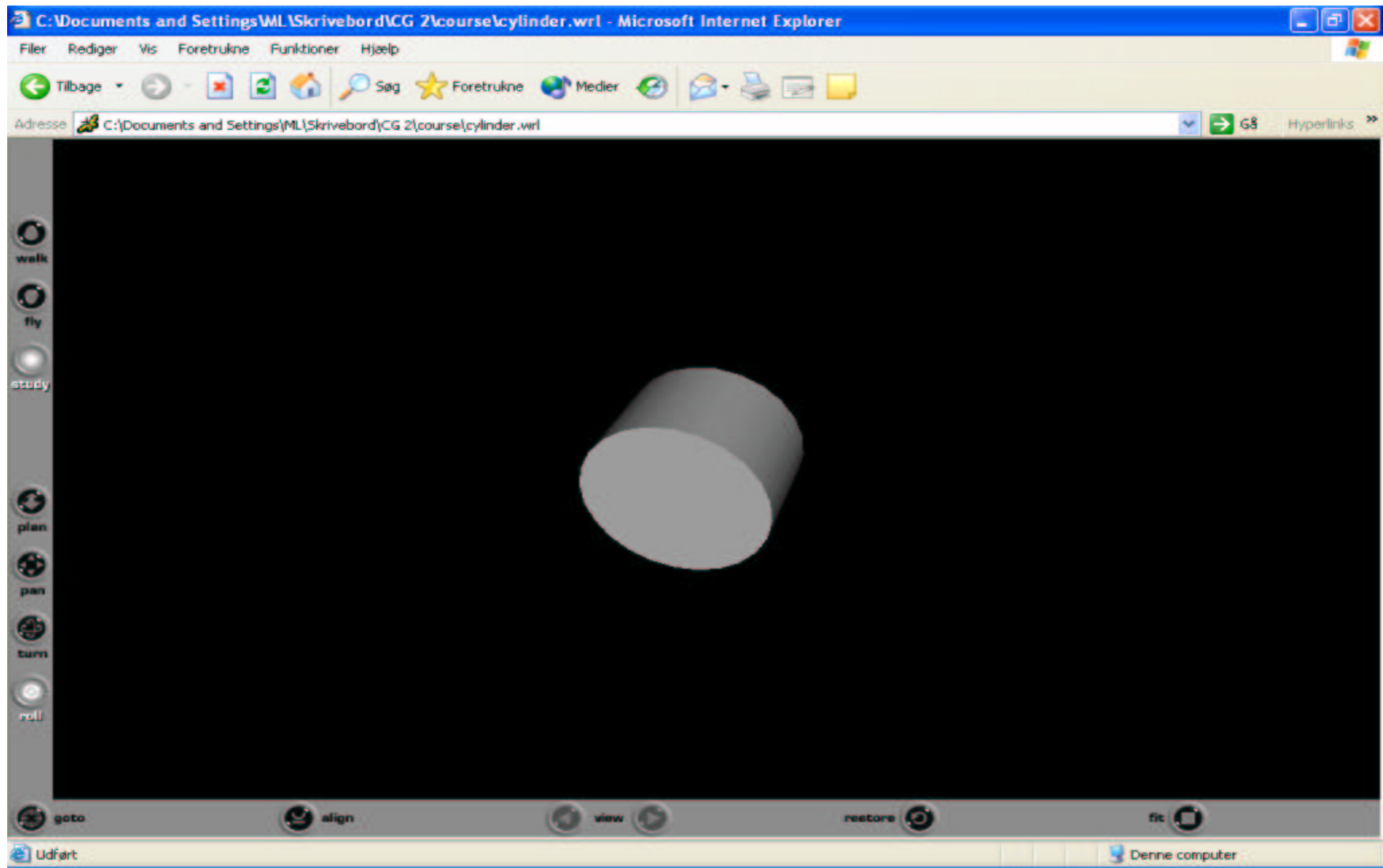
- Geometry node fields control dimensions
 - Box { size 2.0 0.5 3.0 }
 - Cone { height 3.0 bottomRadius 0.75 }
 - Cylinder { height 2.0 radius 1.5 }
 - Sphere { radius 1.0 }



A Sample VRML File

```
#VRML V2.0 utf8
# A Cylinder
Shape {
  appearance Appearance {
    material Material { }
  }
  geometry Cylinder {
    height 2.0 radius 1.5
  }
}
```



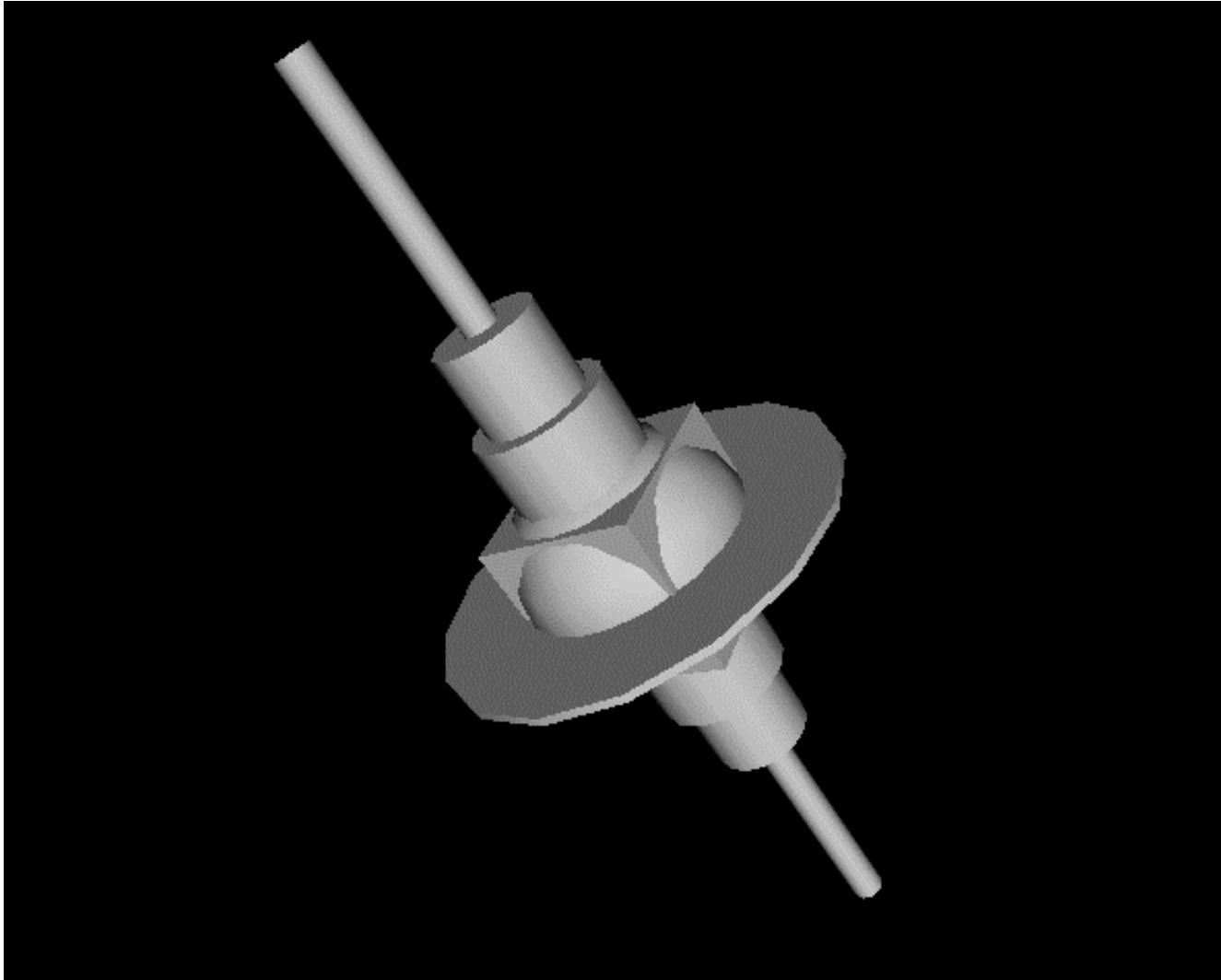


Group nodes

- Group nodes describe:
 - a list of member nodes for the group - *children*

```
Group {  
  children [  
    Shape { . . . },  
    Shape { . . . },  
  ]  
}
```





```
#VRML V2.0 utf8

Group {
  children [
    Shape {
      appearance DEF White Appearance {
        material Material { }
      }
      geometry Box {size 10.0 10.0 10.0}
    },
    Shape {
      appearance USE White
      geometry Sphere {radius 7.0}
    },
    Shape {
      appearance USE White
      geometry Cylinder {radius 12.5 height 0.5}
    },
    Shape {
      appearance USE White
      geometry Cylinder {radius 4.0 height 20.0}
    },
    Shape {
      appearance USE White
      geometry Cylinder {radius 3.0 height 30.0}
    },
    Shape {
      appearance USE White
      geometry Cylinder {radius 1.0 height 60.0}
    }
  ]
}
```



Text Shapes

- Text Shapes build text in a VRML world
 - Signs, notes, annotation, control panels
- Text shapes are flat and have no thickness
- You can choose the font family, style, size, etc.



Text geometry nodes

- Text geometry nodes describe:
 - *text strings* - the text to build
 - *a font style* - how to build the text
 - more . . .

```
Shape {  
  geometry Text {  
    string      . . .  
    fontStyle   . . .  
  }  
}
```



Text & Font styles

- Each text string is built on its own line or column
- Font styles have:
 - *family* - serif, sans, or typewriter
 - *style* - bold, italic, both, or none
 - *size* and *spacing*- character size and spacing
 - *justification* - begin, middle, end
 - *direction* - left/right, up/down



Text Output



A Sample VRML File

```
#VRML V2.0 utf8
Shape {
  geometry Text {
    string [ "SIGGRAPH 96",
            "Conference" ]
    fontStyle FontStyle {
      style "BOLD"
    }
  }
}
```



Transforming Shapes

- By default, all shapes are built at the center of the world
- A *transform* enables you to
 - Position shapes
 - Rotate shapes
 - Scale shapes



Transform group node

- A Transform group node controls:
 - *translation* - position
 - *rotation* - orientation
 - *scale* - size

```
Transform {  
  translation . . .  
  rotation . . .  
  scale . . .  
  children [ . . . ]  
}
```



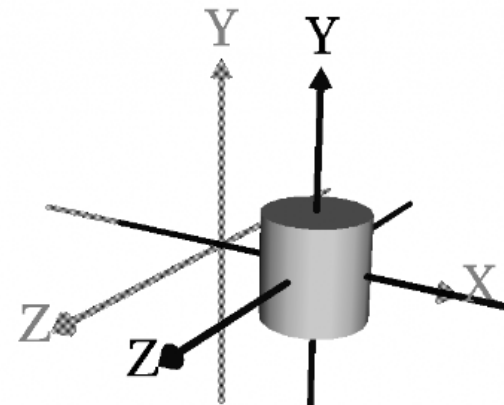
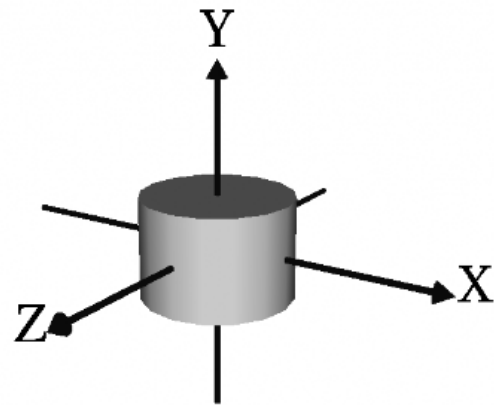
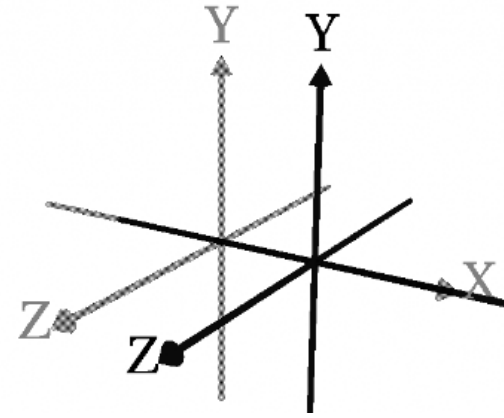
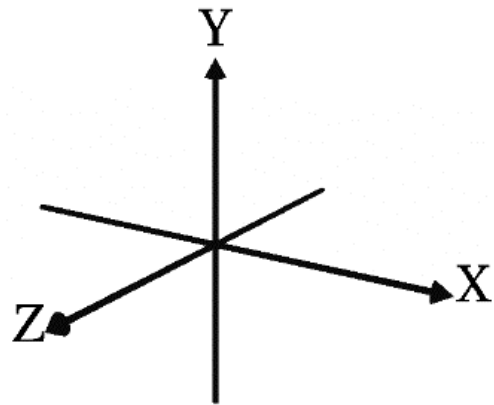
Translating A Coordinate System

- *Translation* positions a coordinate system in X, Y, and Z

```
Transform {  
    #           X   Y   Z  
    translation 2.0 0.0 0.0  
    children [ . . . ]  
}
```



The translation



Rotating A Coordinate System

- Rotation orients a coordinate system about a rotation axis by a rotation angle (in radians)

```
Transform {  
    #          X  Y  Z  Angle  
    rotation 0.0 0.0 1.0 0.52  
    children [ . . . ]  
}
```



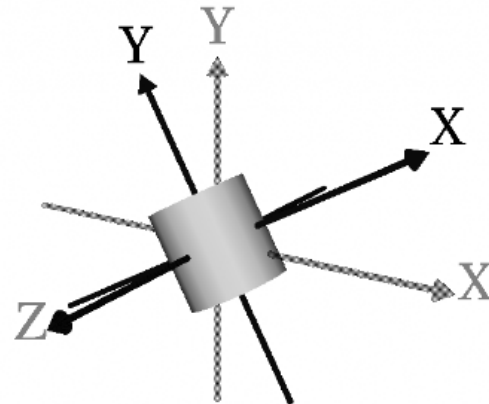
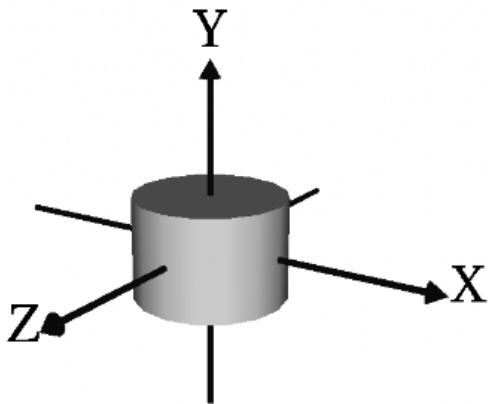
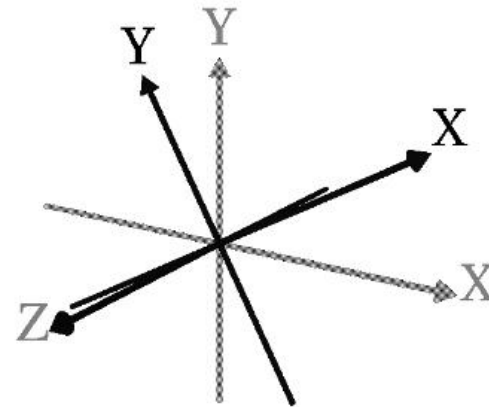
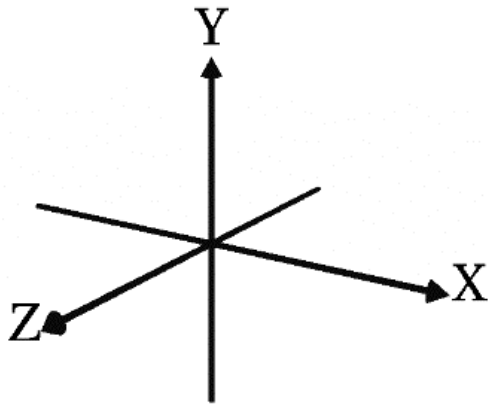
Rotation Axes

- A rotation axis defines a pole to rotate around
- Typical rotations are about the X, Y, or Z axes:

Rotate about	Axis
X-Axis	1.0 0.0 0.0
Y-Axis	0.0 1.0 0.0
Z-Axis	0.0 0.0 1.0



The rotation



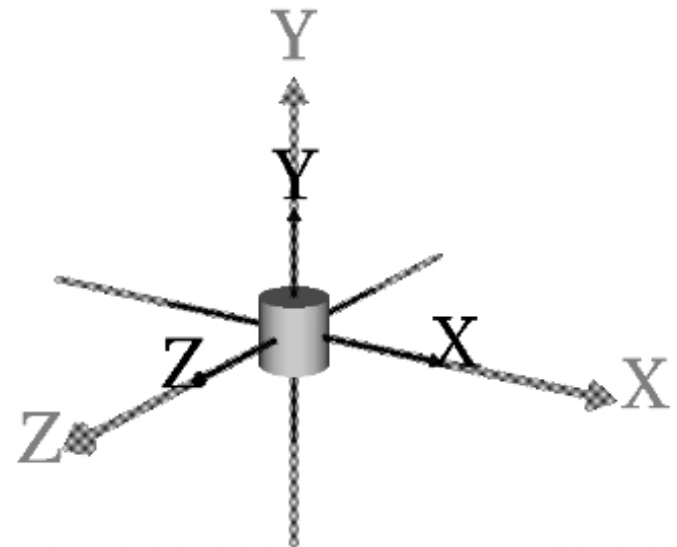
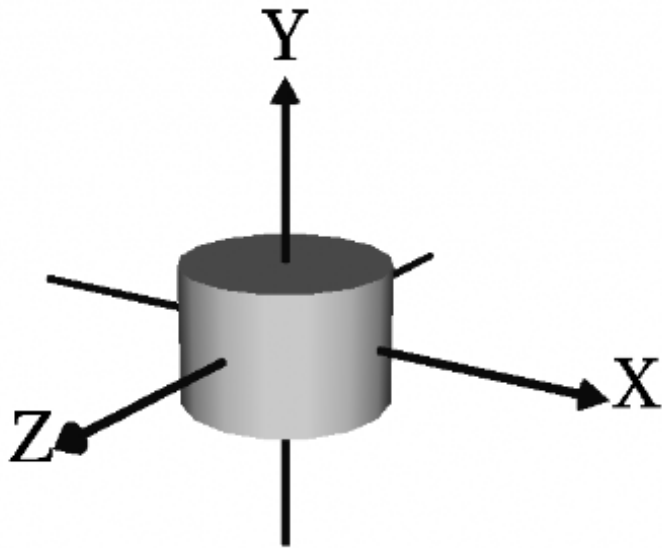
Scaling A Coordinate System

- Scale grows or shrinks a coordinate system by a scaling factor in X, Y, and Z

```
Transform {  
    #      X  Y  Z  
    scale 0.5 0.5 0.5  
    children [ . . . ]  
}
```



The scaling



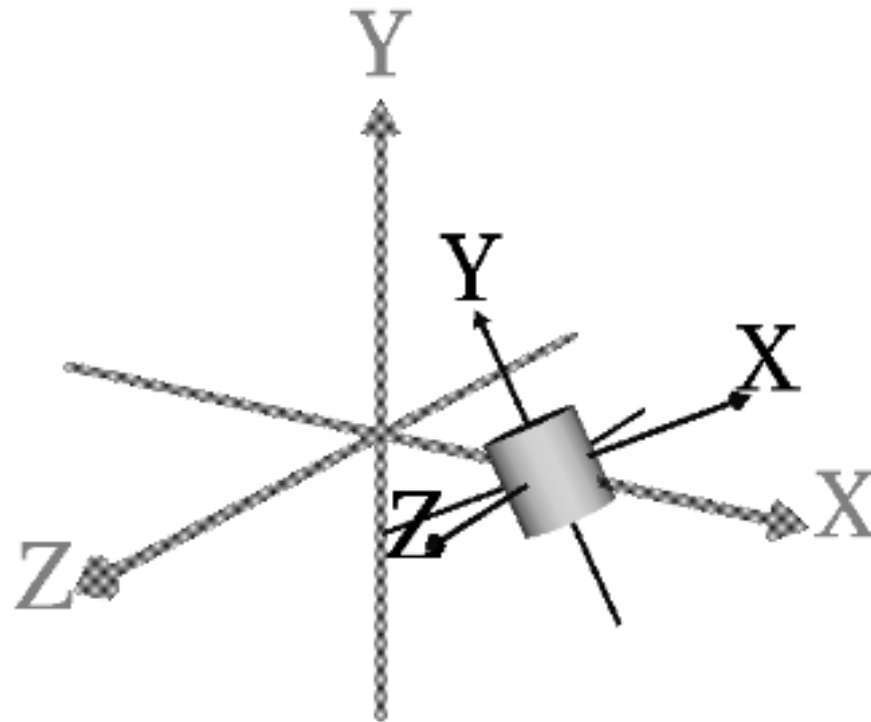
Combining scale, rotate and translate

- *Scale, Rotate, and Translate* a coordinate system

```
Transform {  
  translation 2.0 0.0 0.0  
  rotation    0.0 0.0 1.0 0.52  
  scale      0.5 0.5 0.5  
  children   [ . . . ]  
}
```



The result

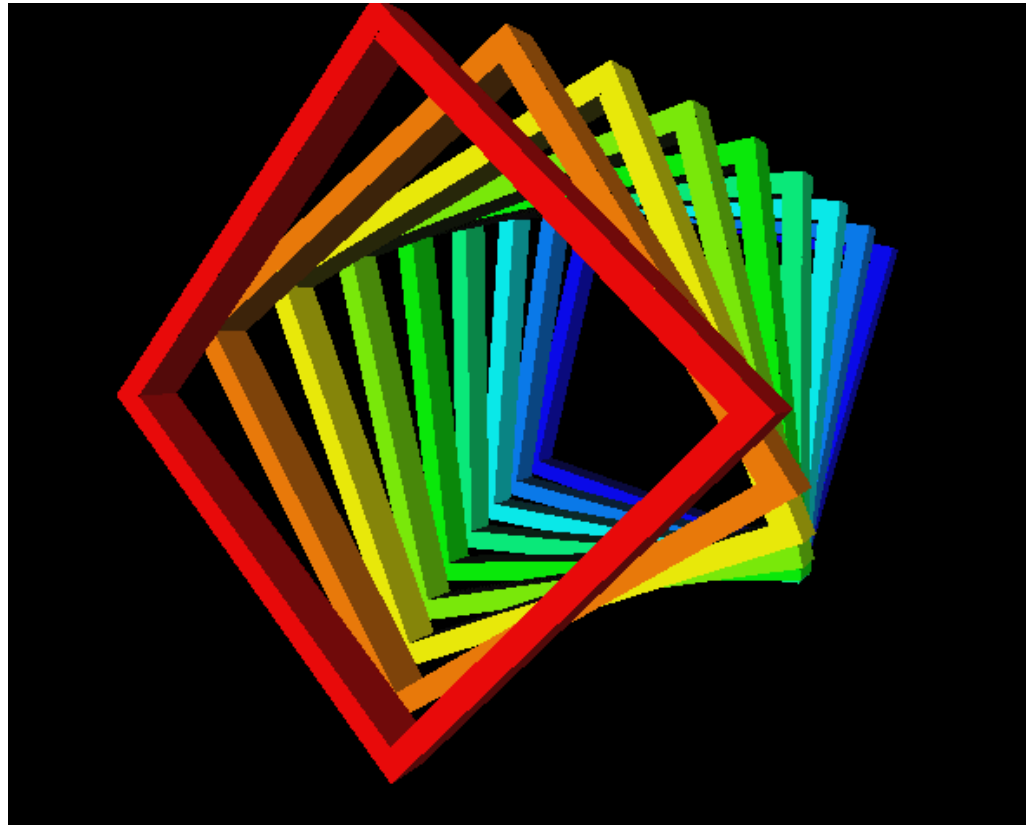


Shape's appearance

- *The primitive shapes have a default glowing white appearance*
- *It is possible to change a shape's*
 - *Shading color*
 - *Glow color*
 - *Transparency*
 - *more . . .*



Color Example



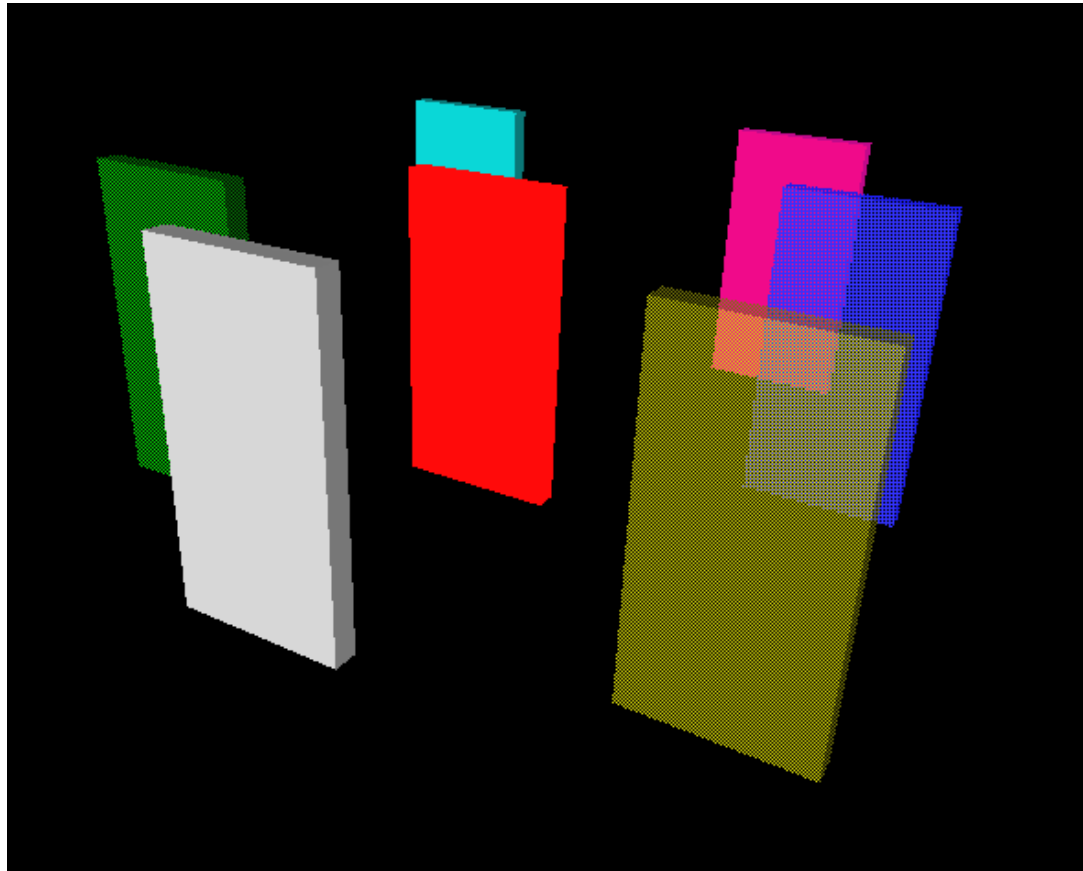
Material node

- A Material node controls:
 - *diffuse color* - main shading color
 - *emissive color* - glowing color
 - *transparency* - opaque or not
 - more . . .

```
Material {  
    diffuseColor    1.0 1.0 1.0  
    emissiveColor   . . .  
    transparency    . . .  
}
```



Changing the color properties



Grouping nodes

- Shapes can be grouped to compose more complex shapes
- VRML has several grouping nodes, including:
 - Group { . . . }
 - Switch { . . . }
 - Transform { . . . }
 - Billboard { . . . }



Group & Switch

- The Group node creates a basic group.
 - *Every child* node in the group is displayed

```
Group {  
    children [ . . . ]  
}
```

- The Switch group node creates a switched group
 - Only *one child* node in the group is displayed
 - You select which child

```
Switch {  
    whichChoice 0  
    choice [ . . . ]  
}
```



Transform & Billboard

- The Transform group node creates a group with its own coordinate system
 - *Every child* node in the group is displayed

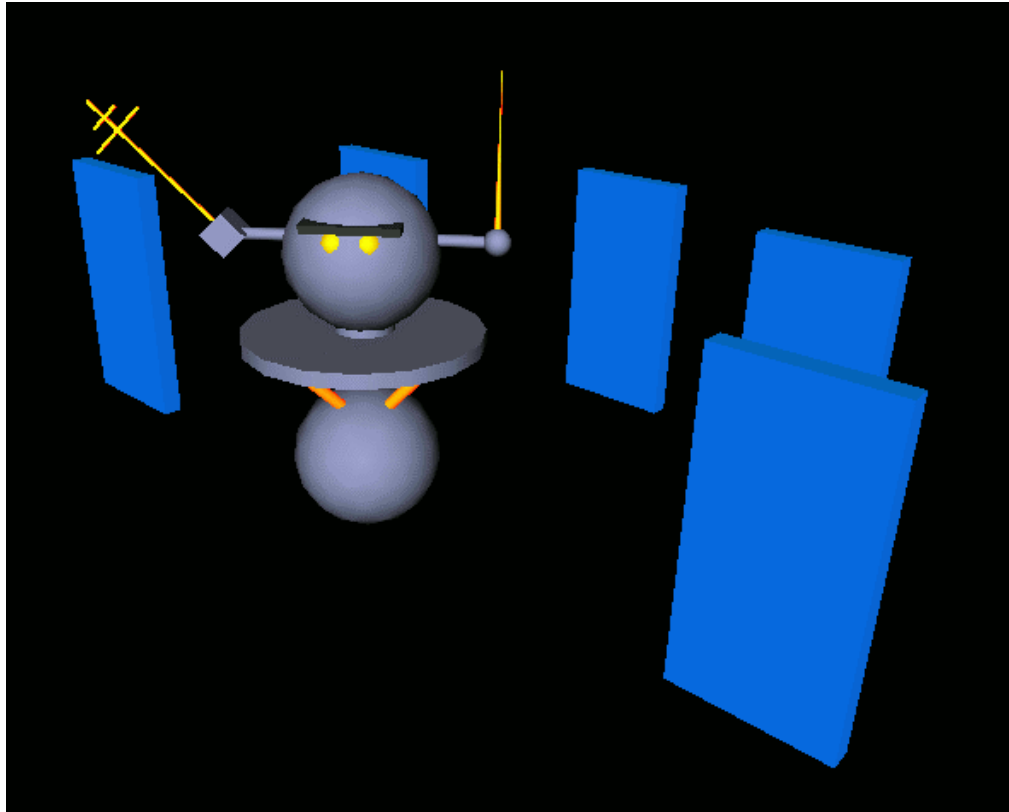
```
Transform {  
    translation ...  
    rotation ...  
    scale ...  
    children [ ... ]  
}
```

- The Billboard group node creates a group with a special coordinate system
 - *Every child* node in the group is displayed
 - Coordinate system is turned to face viewer

```
Billboard {  
    axisOfRotation ...  
    children [ ... ]  
}
```



Robobill.wrl



Inlining World Components

- VRML files describe components of a world
 - Tables, chairs, lamps, walls, floors, doors
- Inlining combines files to build larger components
 - Rooms, buildings, neighborhoods



Using inlining



Inlining source code

```
#VRML V2.0 utf8

Group {
  children [

#   Table

        Inline { url "table.wrl" },

#   Chairs

        Transform {
          translation 0.95 0.0 0.0
          children DEF Chair Inline { url "chair.wrl" }
        },
        Transform {
          translation -0.95 0.0 0.0
          rotation 0.0 1.0 0.0 3.14
          children USE Chair
        },

  ]
}
```



Using Anchors

- The Anchor node creates a group
 - *Every child* node in the group is displayed
 - Selecting any child jumps to VRML file specified by URL
 - The anchor can have a *description* string

```
Anchor {  
    url "stairway.wrl"  
    description "Floating Stairs"  
    children [ . . . ]  
}
```



Building Shapes out of Points, Lines, and Faces

- Create more complex shapes than primitives
- Have more control and flexibility than primitives
- Describe geometry in two steps:
 - Define location of "dots"
 - Connect the dots



Coordinate nodes

- Location of dots (points) are set using a Coordinate node

```
Coordinate {  
  point [  
    2.0 1.0 3.0,  
    4.0 2.5 5.3,  
    . . .  
  ]  
}
```



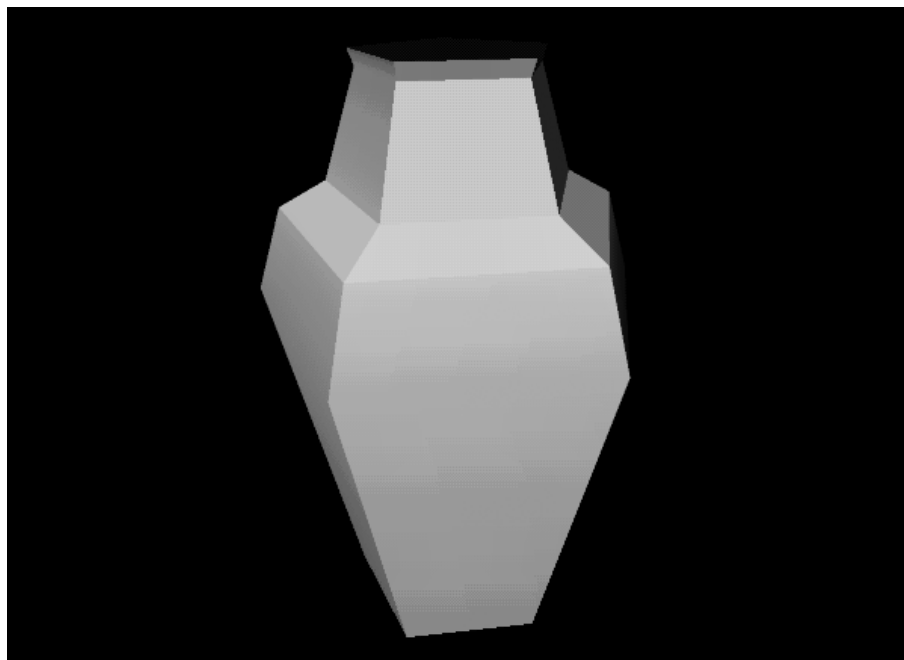
Points, Lines, and Faces

- Three nodes have a coord field which accepts a Coordinate node
 - PointSet
 - IndexedLineSet
 - IndexedFaceSet



IndexedFaceSet

```
IndexedFaceSet {  
  coord Coordinate {  
    point [ . . . ]  
  }  
  coordIndex [  
    1, 0, 3, -1, . . .  
  ]  
}
```



Vase1.wrl



Lights

- Light sources illuminate and effect shading of faces
- Lights do not cast shadows
- Several types of lights available for maximum control
 - PointLight
 - DirectionalLight
 - SpotLight



Lights Standard fields

- Standard fields (common to all 3) and defaults

on TRUE

intensity 1.0

ambientIntensity 0.0

color 1.0 1.0 1.0



Light settings

SpotLight {

Standard fields, plus:

location 0.0 0.0 0.0

direction 1.0 0.0 0.0

beamWidth 1.57

cutOffAngle 0.785

radius 1.0

attenuation 1.0 0.0 0.0

}

PointLight {

Standard fields, plus:

location 0.0 0.0 0.0

radius 1.0

attenuation 1.0 0.0 0.0

}

DirectionalLight {

Standard fields, plus:

direction 1.0 0.0 0.0

}



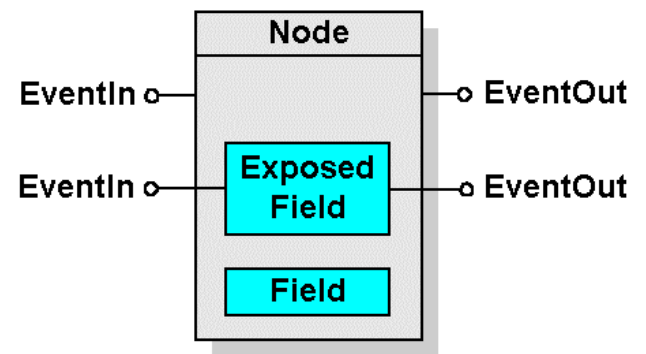
Events

- Event: A message that contains a data value
- Has a specific data type, eg. SFTIME
- set_event: Changes node when received
 - Example: set_startTime
- _changed event: Sent when node changes
 - Example: position_changed



Fields and Events

- Each node has specific fields and events
 - Field: A stored value (parameter), eg. url
 - EventIn: An event node can receive, eg. set_url
 - EventOut: An event node can send, eg. url_changed
- Note: ExposedField defines all three



Route

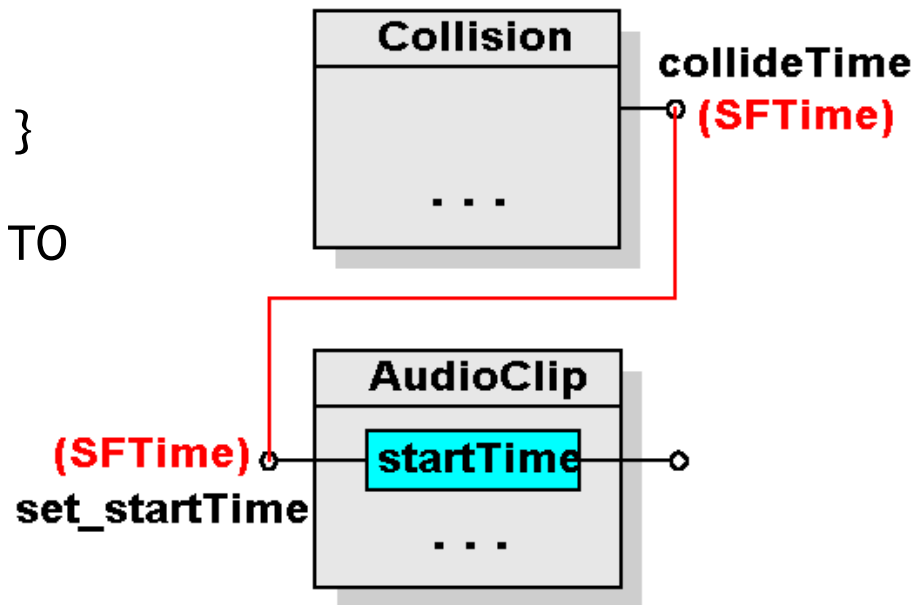
- Route: Connects an EventOut to an EventIn
 - A Route is not a node!
 - Output type must match input type exactly
 - Output may connect to multiple inputs (fan-out)
 - Multiple outputs may connect to single input (fan-in)
- Nodes must be named in order to use Routes



Example: Collision Triggers Sound

```
DEF NODE1 Collision { . . . }  
  
DEF NODE2 AudioClip { . . . }  
  
ROUTE NODE1.CollideTime TO  
NODE2.startTime
```

DEF used to name nodes



M_route.wrl

```
Transform {
  children [
    Sound {
      source DEF Bonk AudioClip {
        url "pop.wav"
        description "You ran into something!"
      },
      location 0 0 0
      minFront 1
      minBack 1
      maxFront 20
      maxBack 5
      spatialize TRUE
    },
    DEF BigThing Collision {
      children [
        DEF CenterPiece Transform {
          ....
        }
      ]
    }
  ]
}

ROUTE BigThing.collideTime TO Bonk.startTime
```



Sensing the Viewer

- Generates events when viewpoint:
 - Enters Region: SFTime enterTime
 - Leaves Region: SFTime exitTime
 - Moves within Region:
 - SFVec3f position_changed
 - SFRotation orientation_changed



The ProximitySensor Node

- `ProximitySensor { center 0 0 0 size 0 0 0 }`
 - Defines an axis-aligned region
 - Zero size disables the sensor
 - All relevant Proximity sensors generate events



Understanding Time

- The sensor generates events while it is running
 - You ROUTE events to change node fields
- TimeSensor nodes describe:
 - *start* and *stop* time - when to run
 - *cycle interval* time - how long a cycle is
 - *looping* - whether or not to repeat cycles

```
TimeSensor {  
    cycleInterval 4.0  
    loop FALSE  
    starTime 0.0  
    stopTime 1.0  
}
```



Timer Input Events

- Can create continuously running timers:
 - loop TRUE
 - stopTime < startTime
- Can run one cycle then stop
 - loop FALSE
 - stopTime < startTime
- Can run until stopped, or after cycle is over
 - loop TRUE or FALSE stopTime >= startTime

