UNIVERSITY OF WATERLOO
Department of Electrical and Computer Engineering

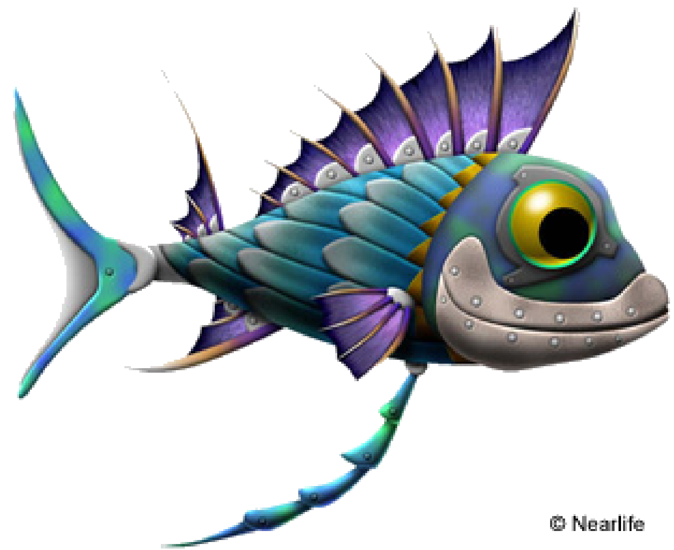
Visualization Using Java3D

Christopher Trudeau
ctrudeau@etude.uwaterloo.ca

Parallel and Distributed Systems Group

Java3D

- create and display virtual worlds
- API uses concepts from VRML
- requires OpenGL for rendering
- scenes can be created through code or by loading VRML or OBJ files
- includes methods for shapes, lighting, behaviours, user interaction, fog model, and detail level



Real World Use

- Division – CAD tools
- Fakespace – virtual world modeling tools
- TempleGames – multi-player interactive games
- NearLife – virtual fishtank at the Boston Computer Museum
- Sun and the VRML Consortium are currently working on a Java3D based VRML browser

Scene Graph

- a scene graph describes everything in the virtual world
- components are asynchronous in order to maximize concurrency
- traversal of the graph is neither depth-first nor breadth-first – renderer decides how to "optimally" traverse the graph
- graph includes:
  - objects – location and grouping of
  - behaviours – periodic (?) events
  - event generation – e.g. collision detection
  - input device handling
- scene graph divided into a *View Branch* and *Content Branch*

Scene Graph Illustration

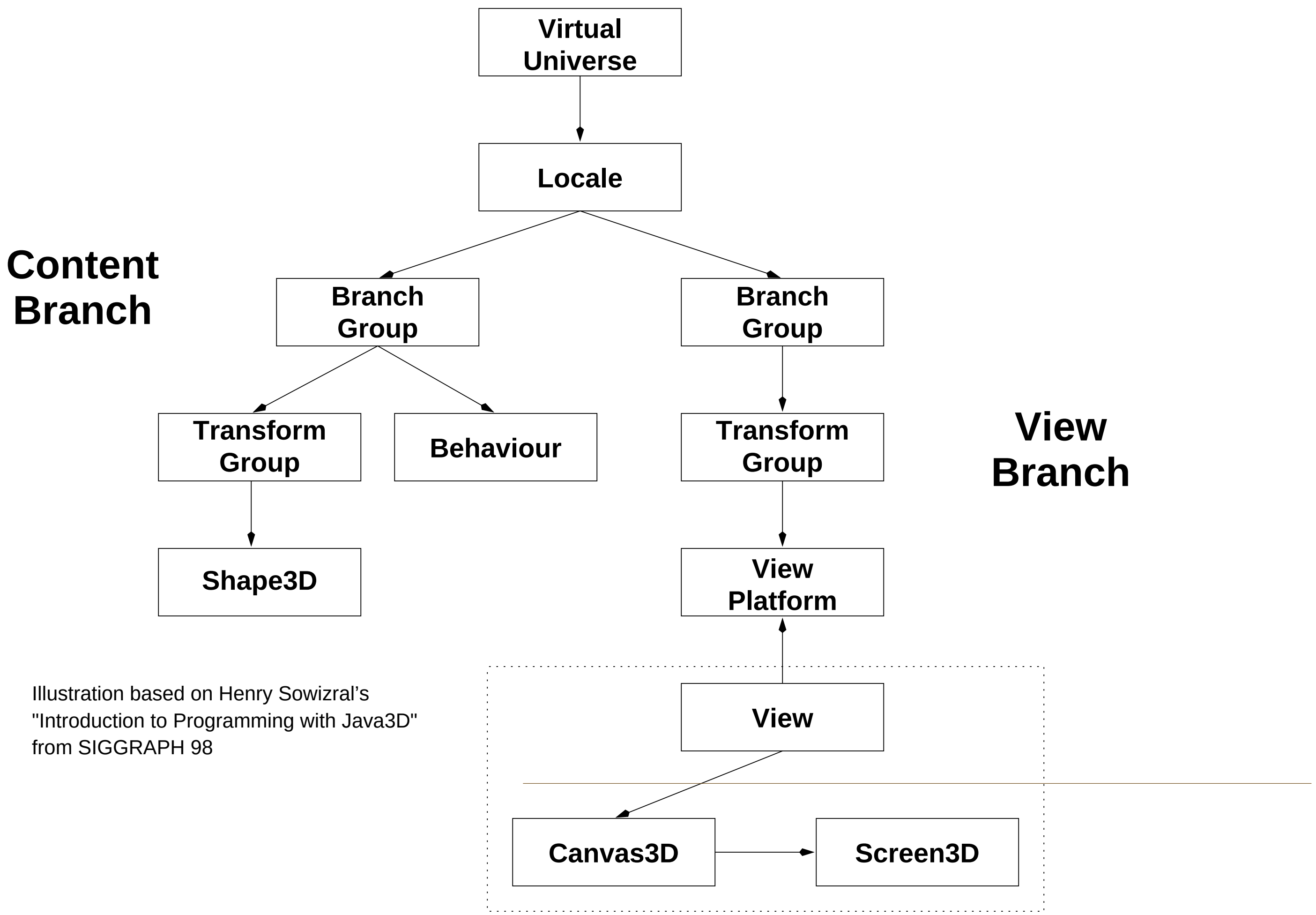


Illustration based on Henry Sowizral's "Introduction to Programming with Java3D" from SIGGRAPH 98

Scene Graph Terms

- *Virtual Universe* – root node, holds the universe
- *Locale* – Java term for geographic location information
- *Branch Group* – generic node used to hold part of the scene graph
- *Transform Group* – all items held under this node are subject to a specific transform
- *Behaviour* – an action which is applied to a transform group; e.g. periodic rotation
- *Shape3D* – shape primitive
- *View Platform* – controls position and scaling of viewer
- *View* – used for viewing a rendered scene; exists outside of scene graph
- *Canvas3D* – an actual viewing object; shows the world; based on Java AWT
- *Screen3D* – single physical viewing device for world

Cosmos – A Simplification

- a lot of the setup for a world is common to different programs
- created Cosmos.java to group this common code in an object

```
private SimpleUniverse universe;
private BranchGroup scene;
private BoundingSphere bounds;

public Cosmos( Canvas3D canvas )
{
    scene = new BranchGroup();       // create a scene graph
    scene.setCapability( BranchGroup.ALLOW_CHILDREN_EXTEND );
    bounds = new BoundingSphere( new Point3d( 0d, 0d, 0d ), 150d );

    // put all of this in a universe
    universe = new SimpleUniverse( canvas );
} // end constructor
```

Let there be light

- Java3D provides a complex lighting model
- includes both ambient and directional light
- light can be any color
- lighting can be for the universe or object specific
- the following code is found in Cosmos' constructor

```
// create some ambient lighting
AmbientLight ambient = new AmbientLight( white );
ambient.setInfluencingBounds( bounds );
scene.addChild( ambient );
addLight( new Vector3f( -10f, -10f, -10f ), white );

public void addLight( Vector3f dir, Color3f color )
{    DirectionalLight lamp = new DirectionalLight( color, dir );
     lamp.setInfluencingBounds( bounds );
     scene.addChild( lamp );
} // end addLight
```

Such Primitive Primitives

- API includes simple objects such as spheres, boxes, test cubes, cones, and cylinders
- use *Appearance* objects to change: colour, transparency, texture mapping, and rendering appearance (e.g. polygon render vs filled render)
- created the Look.java to extend and provide basic appearance manipulation

```
SlickSphere s = new SlickSphere( 2f );
s.look.setTexture( "../images/earth.gif", this );
s.look.setTransparency( 0.5f );
s.create();

class SlickSphere extends Sphere
{   Look look = new Look();

    public SlickSphere( float radius )
    { super( radius, GENERATE_NORMALS | GENERATE_TEXTURE_COORDS, 30 ); }

    public void create() { setAppearance( look ); }
} // end class SlickSphere
```

Transforms

- a *TransformGroup* operates a transformation on all of the nodes contained under it
- created Mover.java class to arbitrarily position an object

```
class Mover extends TransformGroup
{   Vector3d position = new Vector3d( 0d, 0d, 0d );

    public Mover( Vector3d position )
    {   setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );
        moveTo( position );
    } // end constructor

    public void moveTo( Vector3d position )
    {   Transform3D  t = new Transform3D();
        this.position = position;
        t.set( position );
        setTransform( t );
    } // end move
} // end class Mover
```

Transmogrification

- Transform3D class handles many types of transformations
- encapsulates a transformation matrix
- some of the methods:
  - determinant()
  - frustum() – type of projection transform
  - invert()
  - mulInvert() – multiply and invert
  - mulTransposeLeft/Right/Both() – multiple and transpose
  - mul() – various multiplications
  - normalize()
  - ortho() – orthographic projection
  - perspective() – perspective projection
  - rotX/Y/Z() – rotate with respect to X/Y/Z axis
  - scale()
  - transform(Object) – transform the object passed in

Behaviour – How do you get the room to spin like this?

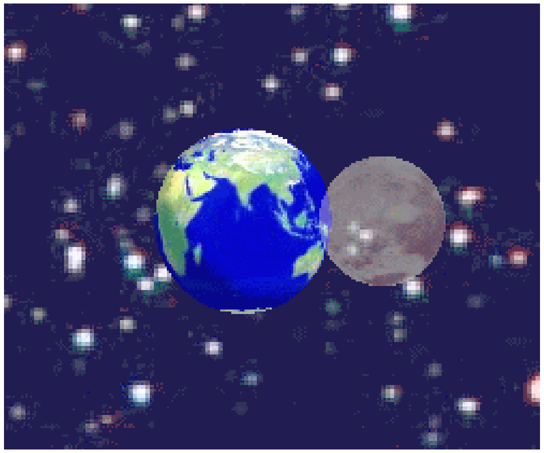
- a behaviour is an "action" which is performed by an object; e.g. rotation (spinning or around something), making noise, etc.
- an *Alpha* object converts time into a value [0,1]
- an *Interpolator* object interpolates between two things
- a *RotationInterpolator* changes a *TransformGroup*'s rotation property by interpolating between two angles

```
public Spinner() // extends TransformGroup
{   setCapability( ALLOW_TRANSFORM_WRITE );
    Transform3D yAxis = new Transform3D();

    Alpha rotationAlpha = new Alpha( -1, Alpha.INCREASING_ENABLE,
        0, 0, 5000, 0, 0, 0, 0, 0 );

    RotationInterpolator rotator = new RotationInterpolator(
        rotationAlpha, this, yAxis, 0f, (float)(2d * Math.PI) );

    addChild( rotator );
}
```

Behaviour – Rotate This

- can use Spinner.java to create rotation or orbit

```
// spinning sphere at centre of universe
Spinner r = new Spinner();
cosmos.addChild( r );
SlickSphere s3 = new SlickSphere();
r.addChild( s3 );

// sphere which rotates around centre
Mover m = new Mover( new Vector3d( 4d,
       0d, 0d ) );
r = new Spinner();
cosmos.addChild( r );
r.addChild( m );
SlickSphere s4 = new SlickSphere( 0.4f );
m.addChild( s4 );
```



Controlling the View

- Java3D treats the view as another part of the scene graph
- this means the view can be transformed
- traditional 3D software translates the world rather than the view
- to "fly" through the world simply translate the view
- controlling the flight is done via a *Behaviour*

```
public SensorBehaviour( TransformGroup tg, Sensor sensor )
{   conditions = new WakeupOnElapsedFrames( 0 );... }

public void initialize() { wakeupOn( conditions ); }
public void processStimulus( Enumeration criteria )
{   sensor.getRead( transform );     // read from the sensor
    transformGroup.setTransform( transform ); // apply transform

    wakeupOn( conditions ); // set the next wake up
} // end processStimulus
```

Sensors and Devices

- SensorBehaviour.java references a *Sensor* which is associated with a *Device*
- ViewControls.java is a six-degree of freedom virtual device
- a collection of buttons allow the user to move in any direction and change their rotation POV
- pushing a position changing button updates a position vector
  - not quite as simple as it sounds
  - e.g. left does not mean just add to the X axis since we want to step left in accordance to the direction we are facing
  - transform X axis addition: multiply the motion vector by current rotational matrices
- pushing a rotational button modifies a rotational transform
- the sensor is polled periodically by the universe
- each time the sensor is polled the sensor's rotation and position is updated from the above calculated information