

Java3D: Environment

- Environment node in the scene graph describe the region in which the objects exist in the virtual universe. We will focus on:
 - Lights
 - Fog
 - Backgrounds
 - Behaviours
- Java3D does not know about the physics of the objects in the virtual universe. To limit the effect on an environment node on the virtual universe we use *bounding regions*.
 - Every environment node must have a bounding region, otherwise Java3D will make the node inactive.
 - Bounding regions also help the Java3D engine maximise its performance.

Bounding Regions

- The basic bounding regions in Java3D are:
 - **BoundingBox** that defined a rectilinear box aligned with the X, Y and Z axes.
 - **BoundingSphere** that defines a sphere at a specified point and with a specified radius.
 - **BoundingPolytope** that defines a set of mathematical planes that enclose a convex region of the virtual universe.

```
BoundingSphere infiniteBounds =  
new BoundingSphere  
(new Point3D(  
    Double.MAX_VALUE);
```

- Defines a infinite bounding region centred on the origin.

Lights

- Java3D supports the following types of lighting.
 - *Directional lights* from a distant source.
 - *Point lights* at a specific point in space.
 - *Spot lights* that are located at a specific point and are pointing in a specific direction.
 - *Ambient lights* that diffusely light an area.
- All the types of lights are derived from a base class **Lights** and have the following methods:

```
void setEnabled (boolean state);
boolean getEnabled ();
void setColor (Color3f colour);
void getColor (Color3f colour);
```
- If a light is to be modified when it is a live node then the appropriate capability bits must be set.
 - ALLOW_STATE_WRITE,
 - ALLOW_STATE_READ,
 - ALLOW_COLOR_WRITE,
 - ALLOW_COLOR_READ etc.

Directional Lights

- These are light that shine in a specified direction not matter where you are in the universe. The light rays are parallel from a particular direction but no particular source.
- Diffuse effects such as colour shading will be the same for all object though the specular effects will depend on the location of the viewer.

```
Vector3f lightDir = new Vector3f (0.6f, -0.6f, -0.4f);
Color3f white = new Color3f (1.0f, 1.0f, 1.0f);
DirectionalLight sun =
    new DirectionalLight (white, lightDir);
sun.setInfluenceBounds (infiniteBounds);
sun.setCapability
    (Light.ALLOW_STATE_WRITE);
```
- If the ALLOW_DIRECTION_WRITE capability is set then the direction of the light can be modified by

```
void setDirection (Vector3f direction);
```

Point Lights

- A **PointLight** is located at a specific location in space but radiates light in all direction.
- The light intensity attenuates with distance from the sources of the light. In the real world this attenuation is an inverse square law.
- In Java3D the attenuation model is:
$$\text{intensity}_d = \text{intensity}_{\text{source}} / (\text{const} + \text{lin} * d + \text{quad} * d^2)$$
- While a quadratic drop-off in light intensity is more accurate, a near-linear drop-off looks more realistic.
- The attenuation factor is specified as a Point3f with the x, y and z components representing the constant, linear and quadratic coefficients respectively.

PointLight Example

```
Point3f lightPos = new Point3f (-1.0f, 1.0f, 2.0f);
Point3f lightAttenuation = new Point3f (0.0f, 0.8f,
0.2f);
Color3f white = new Color3f (1.0f, 1.0f, 1.0f);
BoundingSphere infiniteBounds =
new BoundingSphere
    (new Point3D( ),
    Double.MAX_VALUE);
PointLight bulb =
    new PointLight (white, lightPos,
    lightAttenuation);
bulb.setInfluenceBounds (infiniteBounds);
bulb.setCapability
    (Light.ALLOW_STATE_WRITE);
group.addChild (bulb);
```

Spot Lights

- Spot lights have a location, a direction and can be focused.
- In Java3D the control of a **SpotLight** object's spot is by a spread angle and concentration.
 - Spread angle determines the maximum width of the light beam.
 - The concentration determines how quickly the light intensity falls off as it moves away from the spot light's direction vector.
 - Concentration = 0 implies no effect
 - Concentration = 128 implies the light shines in a narrow beam along the direction vector.
- **SpotLight** also has an attenuation factor.

SpotLight Example

```
Point3f lightPos = new Point3f (-1.0f, 1.0f, 2.0f);
Point3f lightAttenuation = new Point3f (0.0f, 0.8f,
0.2f);
Vector3f lightDir = new Vector3f (0.6f, -0.6f, -0.4f);
float spotConcentration = 128.0f;
float spreadAngle = 60.0; // degrees

Color3f white = new Color3f (1.0f, 1.0f, 1.0f);

BoundingSphere infiniteBounds =
new BoundingSphere
(new Point3D( ),
Double.MAX_VALUE);

PointLight spot =
new SpotLight (white, lightPos,
lightAttenuation, lightDir,
(float) Math.toRadians
(spreadAngle),
spotConcentration);

bulb.setInfluenceBounds (infiniteBounds);

group.addChild (spot);
```

Ambient Lights

- Ambient lights have no source or direction. They model light that has been reflected of multiple surfaces until it completely illuminates a space.

```
Point3f[] sceneVertex = new Point3f [5];
sceneVertex [0] = new Point3f (1.0f, 0.0f, 1.0f);
sceneVertex [1] = new Point3f (-1.0f, 0.0f, 1.0f);
sceneVertex [2] = new Point3f (-1.0f, 0.0f, -1.0f);
sceneVertex [3] = new Point3f (1.0f, 0.0f, -1.0f);
sceneVertex [4] = new Point3f (0.0f, 1.0f, 0.0f);
Bounds sceneBounds = new Bounds ( );
sceneBounds.combine (sceneVertex);
```

```
Color3f grey = new Color3f (0.2f, 0.2f, 0.2f);
```

```
AmbientLight ambLight =
    new AmbientLight (grey);
```

```
ambLight.setInfluentBounds (sceneBounds);
group.addChild (ambLight);
```

Fog

- Fog causes objects at a distance to fade; the further away from the viewer the more pronounced in the effect. This “depth cueing”, in conjunction with perspective, gives the brain a visual clue to the object’s depth in a scene.
- Java3D support two types of fog:
 - **LinearFog**
 - Has 3 parameters; colour, start distance (fog starts) and end distance (fog complete)
 - **ExponentialFog**
 - approximates real world fog.
 - Has 2 parameters; colour and exponential decay factor.

Fog Example

```
Color3f skyBlue = new Color3f (0.6f, 0.7f, 0.9f);
```

```
LinearFog linFog =  
    new LinearFog  
        (skyBlue, 7.5f, 15.0f);  
linFog.setInfluenceBounds  
    (infiniteBounds);
```

```
fogSwitch.addChild (linFog)
```

```
ExponentialFog expFog = new  
ExponentialFog (skyBlue, 0.3f);  
expFog.setInfluenceBounds  
    (infiniteBounds);
```

```
fogSwitch.addChild (expFog);
```

Backgrounds

- Backgrounds specify the visual content that appears behind the scene.
- A **Background** could be a colour.

```
Color3f skyBlue = new Color3f (0.6f, 0.7f, 0.9f);  
Background bgColour = new Background  
    (skyBlue);  
bgColour.setApplicationBounds (infiniteBounds);  
bgSwitch.addChild (bgColour);
```

- A **Background** can be an image, loaded using the **TextureLoader**.

```
URL bgImageURL;  
TextureLoader bgTexture = new TextureLoader  
    (bgImageURL, null);  
Background bgImage = new Background  
    (bgTexture.getImage ());  
bgImage.setApplicationBounds (infiniteBounds);  
bgSwitch.addChild (bgImage);
```

Backgrounds (contd.)

- Unless the **Background** has a geometry it will not change as the viewer moves around the virtual universe.

```
URL imageUrl;  
TextureLoader bgTexture =  
    new TextureLoader (imageUrl, null);
```

```
Background bgGeo = new Background ();  
bgGeo.setApplicationBounds (bounds);  
BranchGroup bgGeoBG = new BranchGroup ();  
Appearance bgGeoApp = new Appearance ();  
bgGeoApp.setTexture (bgTexture.getTexture ());
```

```
Sphere sphereObj = new Sphere (1.0f,  
    Sphere.GENERATE_NORMALS |  
    Sphere.GENERATE_NORMALS_INWARDS |  
    Sphere.GENERATE_TEXTURE_COORDS, 45,  
    bgGeoApp);
```

```
bgGeoBG.addChild (sphereObj);  
bgGeo.setGeometry (bgGeoBG);  
bgSwitch.addChild (bgGeo);
```

Behaviours

- **Behavior** nodes change the scene graph in response to an event.
 - User input
 - Passing of time
- Java3D has several predefined **Behavior** nodes and also allows the user to define new **Behavior** nodes.
- The **Behavior** class define 2 methods:
 - void `setEnabled` (boolean state);
 - Whether the behaviour is enabled.
 - void `setSchedulingBounds` (Bounds regions);
 - Defines the regions in which the behaviour is active.

Predefined Behaviours

- DistanceLOD
 - Controls a **Switch** node to provide different levels of detail (LOD). The distance from the **ViewPlatform** controls which child of the **Switch** to render.
- Billboard
 - Controls a **TransformGroup** to keep its children oriented towards the **ViewPlatform**.
- Interpolator
 - Changes a transform or attribute over time based on an **Alpha** value. The constructor of **Alpha** takes two values:
 - Loop count: -1 means loop infinitely
 - Loop duration in milliseconds.
 - Simple Interpolators go from a start to end value. Path Interpolators go through a series of values (knots) where the first knot corresponds to Alpha=0 and the last knot corresponds to Alpha=1.

Java3D: Transforms

- Transforms are applied to **TransformGroup** nodes in order to translate, scale and rotate the children of the **TransformGroup**. To apply a transform to a **TransformGroup** simply call the method:

```
void setTransform  
    (Transform3D transform);
```

- This method copies the **Transform3D** to the **TransformGroup**. Subsequent changes to the **Transform3D** does not change the **TransformGroup**, until `setTransform()` is called again.

Translation

- A translation adds an offset to the x, y and z coordinates. This offset is defined by a **Vector3f**.

```
Vector3f tmpVector = new Vector3f ();  
tmpVector.set (1.0f, 0.0f, 0.0f);  
Transform3D tmpTrans =  
    new Transform3D ();  
tmpTrans.set (tmpVector);  
shapeTG.setTransform (tmpTrans);
```

- For an existing shape:

```
void setShapeTranslation (Vector3f  
shapeTranslation)  
{  
    Transform3D tmpTrans;  
    shapeTG.getTransform (tmpTrans);  
    tmpTrans.setTranslation  
        (shapeTranslation);  
    shapeTG.setTransform (tmpTrans);  
}
```

Scaling

- Scaling can be:
 - Uniform, applying equally to all dimensions.
void set (double scale);
 - set() sets the scale but resets the translation and rotation.
 - void setScale (double scale);
 - setScale() only effects the scale elements of the transform.
 - Nonuniform with different scaling factors applied to each dimension.

```
void setScale (Vector3f scale);

- The x, y and z components of the scale vector represent the x, y and z scale factors.

```

Rotation

- A rotation is defined by an axis to rotate around (defined by a vector) and an angle. The direction of the rotation is determined by the “Right Hand Rule”.
 - Align your right thumb along the axis of rotation and when you wrap the rest of your finger around the axis, your fingers point in the direction of positive rotation.
- The methods are:
void set (AxisAngle4f aa);
void setRotation (AxisAngle4f aa);

where **AxisAngle4f** has constructors:

AxisAngle4f (float x, float y, float z,
float angle);

AxisAngle4f (Vector3f axis, float angle);

where angle is expressed in radians.

Composing Transforms

- All transforms are represented as matrices. The order in which transforms is *very important*.
 - Rotating 90° about x-axis, translating 2 units along the x-axis and rotating 90° around y-axis is **not the same as** rotating 90° about x-axis, rotating 90° around y-axis and translating 2 units along the x-axis.
- To compose transforms to represent applying A then B and then C multiply the matrices in the following order.

$$T = C * B * A$$

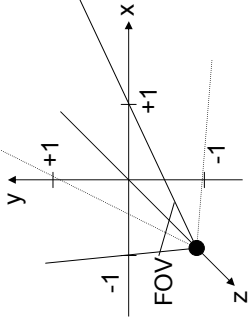
- Use the methods
void mul (Transform3D t1, Transform3D t2);
 - this = t1*t2void mul (Transform3D t1);
 - this = this*t1

Java3D: Viewing

- In Java3D the **ViewPlatform** represents the viewer and the **View** maps the viewpoint to a 2D image in a **Canvas3D**.
- The *field of view* (FOV) of the **ViewPlatform** determines how much of the scene appears in the **Canvas3D**.
- If the **ViewPlatform** is placed in a **TransformGroup** then it can be moved around the world and change the rendered image.
- The default projection used by **View** is *perspective projection*.

Basic Viewing

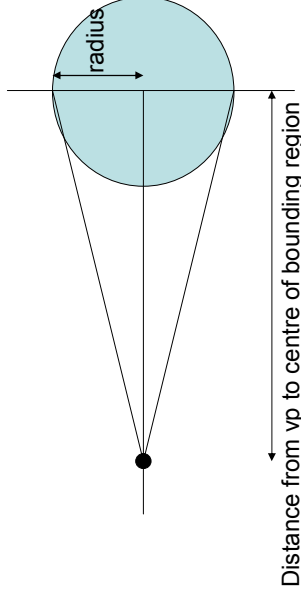
- In the HelloUniverse program we used the following method.

```
u.getViewingPlatform().setNominalViewingTransform();
```
 - `setNominalViewingTransform()` moves the **ViewPlatform** so that the range $[-1, 1]$ is visible in the X and Y axes at $z = 0$.
- 
- The default FOV is 45° . This can be changed by the **View** method:

```
void setFieldOfView (double fov);
```

Moving the ViewPlatform

- For scenes that don't fit into $[(-1, -1), (1, 1)]$ we need to figure out where to place the **ViewPlatform**. One technique is to use the scene's **BoundingSphere**.



$$\text{viewDistance} = 1.4 * \text{radius} / (\tan(\text{FOV}/2))$$

- Use the 1.4 factor to be safe.

Setting ViewPlatform Example

```
sceneGroup.setCapability
(BranchGroup.ALLOW_BOUNDS_READ);
liveGroup.add(sceneGroup); // make scene live
BoundingSphere sceneBounds =
(BoundingSphere) sceneGroup.getBounds ();
double radius = sceneBounds.getRadius();
Point3d centre = new Point3d ();
sceneBounds.getCenter (centre);

SimpleUniverse u;
View view = u.getViewer().getView();

Vector3d viewVector = new Vector3d (centre);

double viewDistance =
1.4 * radius / (Math.tan (view.getFieldOfView() / 2.0));
viewVector.z += viewDistance;

Transform3D viewTransform = new Transform3D ();
viewTransform.set (viewVector);
u.getViewPlatformTransform().setTransform(viewTransform);
```