Nectar3D: A 3D Scene Graph Engine in Java

by Lindsay Kay

December 2005

Abstract

Nectar3D is a simple 3-D scene graph engine in pure Java, featuring shading, depth-cueing and keyframe animation. The Nectar3D renderer fires events when scene elements are picked or moused-over, and may be instructed to highlight selected sets of scene elements. The scene graph can be dynamically modified "at runtime", where elements may be added, deleted or moved while the renderer continually renders it.

Contents

1 Overview

A Nectar3D scene graph specifies the appearance and behaviour of a 3D world. Elements within a scene can be illuminated with multiple light sources (shaded), made to appear dimmer as they recede into the distance (depth-cued), rendered in order of depth to resolve visual overlap (depth-sorted), and animated through interpolation of their positions, sizes and orientations within keyframe sequences. The graph can also be partitioned into *layers* which are rendered seperately in the order specified, typically with different rendering features enabled.

The scene graph is mutable, where elements may be created, destroyed, moved and updated at any time.

A convenient SceneBuilder is provided with which to construct a scene graph, and a SceneRenderer is provided with which to render it.

Each subtree within a scene can be named so that when an element contained within it is selected from a rendered image (by intersection with given 2-D screen coordinates) the SceneRenderer can fire a selection event identifying the subtree. A set of subtree names can also be specified to the SceneRenderer so that when it renders a scene it will highlight elements within those subtrees.

The following sections describe the scene graph, SceneBuilder and SceneRenderer.

2 The Scene Graph

The scene graph is a tree structure over which a SceneRenderer traverses in depth-first, left-to-right order.

2.1 Element Types

The root element of a scene graph is a `SceneElement`, while all other elements are sub-classes of this type. There are eight types of non-root element:

- an `Environment` contains light sources to illuminate sub-elements,
- a `Layer` indicates to a `SceneRenderer` which features it should apply when rendering sub-elements,
- a `TransformGroup` specifies transformations to apply to sub-elements,
- an `Interpolator` animates a transformation within a parent `TransformGroup`,
- a `Geometry` contains `Point3`s and `Face`s,
- a `Label` is an element of text oriented towards the viewpoint,
- an `Appearance` specifies the colour(s) of it's parent, and
- a `Name` defines a collective name for sub-elements.

These elements are described in more detail in the following sections.

2.1.1 Environment

An `Environment` contains one or more `LightSource`s, each of which specifies the direction and colour of a light source which illuminates sub-elements.

Each `LightSource` has a direction `Vector3` and a `Color`. The Nectar3D shading model is Lambertian (flat), where each rendered `Geometry Face` is uniformly filled by a colour which is a mix of the `Face`'s original colour and the colours of light reflected from all `LightSource`s.

Note that the intensity of a `LightSource` does not attenuate with distance (perhaps this is a feature for the next Nectar3D version).

Although an `Environment` can be the root of a scene graph, it is typically a child of the root `SceneElement`, with other `Enviroments` as siblings. The only type of element that can be a child of a `Environment` is a `Layer`.

2.1.2 Layer

A `Layer` specifies which features should be applied by a SceneRenderer when it renders the `Layer`'s sub-elements (see Section **??**).

To see how `Layer`s work, consider the example scene graph of Figure **??**, which contains objects against a backdrop of sky and ground. This scene has a `Layer` containing ground and sky as two `Geometry` sub-elements, perhaps a large green polygon and a large blue polygon. Depth-sorting and shading **are not** to be applied by a `SceneRenderer` for that `Layer`. The second `Layer` contains an animated `Geometry` element for a foreground object, and for that `Layer` those two rendering techniques **are** to applied by a `SceneRenderer`. Since a `SceneRenderer` traverses a scene graph from left to right, the first `Layer` would be rendered first with the second rendered on top. Note that fogging is not applied to sub-elements of the foreground `Layer` since fogging blends each element's colours with the scene's background colour (configured with a `SceneRendererParams` as described in Section **??**), which is obscured by the background `Layer` in this example.

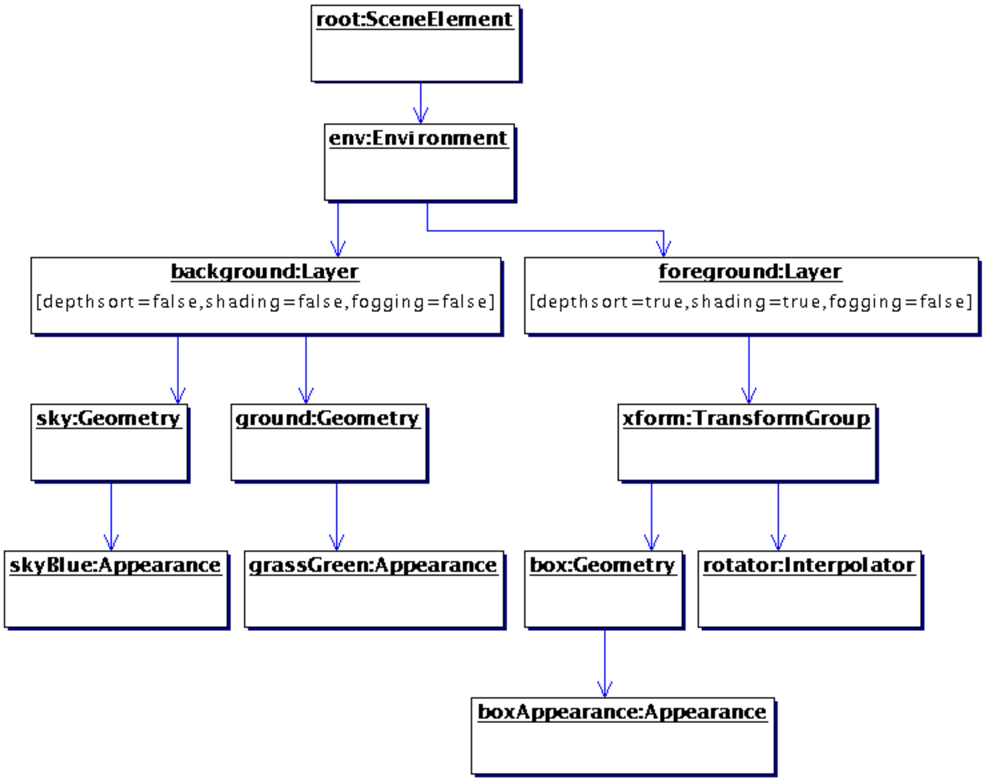

Figure 1: UML instance diagram of a scene graph containing background and foreground `Layer`s. Note how the background layer specifies to a `SceneRenderer` that it should **not** apply features such as depthsort and shading when rendering it's sub-elements.

The elements that can be direct children of a `Layer` are:

- `Geometry`,
- `TransformGroup`,
- `Label`,
- `Box`, and
- `Name`.

2.1.3 TransformGroup

A `TransformGroup` defines a sequence of affine transformations to apply to sub-elements. As described in Sections ?? and ??, the SceneRenderer renders elements such as `Geometry` and `Label` after first transforming them by each `TransformGroup` on the path from the element up to the root.

Affine transformations are added to a `TransformGroup` one at a time with the element's `addTransform` method. An element transformed by a `TransformGroup` is transformed by each of the transformations in the order in which the transformations were added.

The following constants are defined by `TransformGroup` to identify the five types of transformations which can be added:

- `SCA` - scale on X, Y and Z-axis,
- `TRA` - translate on X, Y and Z-axis,
- `ROT_X` - rotate about X-axis,
- `ROT_Y` - rotate about Y-axis, and
- `ROT_Z` - rotate about Z-axis.

Each type of transformation can only be added once to a `TransformGroup`.

Scale factors, translation offsets and rotation angles can be individually set and got for each axis with methods `setAttribute` and `getAttribute`. The following constants identify those attributes:

- `SCA_X`, `SCA_Y` and `SCA_Z` identify factors by which to scale on X, Y and Z-axis,
- `TRA_X`, `TRA_Y` and `TRA_Z` identify offsets by which to translate on X, Y and Z-axis, and
- `ROT_X`, `ROT_Y` and `ROT_Z` identify angles in degrees about which to rotate on X,Y and Z-axis.

The elements that can be direct children of this element are:

- `Geometry`,
- `TransformGroup`,
- `Layer`,
- `Interpolator`, and
- `Name`.

As described in Section ??, one or more `Interpolator`s can be connected as children of a `TransformGroup` to animate individual transformations.

2.1.4 Interpolator

An `Interpolator` animates a single affine transformation within a parent `TransformGroup`. An `Interpolator` contains a series of keyframes, each of which specifies a value for the transformation's attribute at an instant in time, given in milliseconds, relative to the instant that the `SceneRenderer` was started.

As described in Section **??**, a `SceneRenderer` tracks the milliseconds elapsed since it was started. When a `SceneRenderer` visits an `Interpolator`, it updates the `Interpolator` with the elapsed time. If the `Interpolator` has two keyframes that enclose the elapsed time it sets the attribute of it's transformation to a value linearly interpolated between the keyframe values as a function of the elapsed time. If no such keyframes exist, the `Interpolator` does nothing. An `Interpolator` can therefore be made to lie dormant for a period by specifying a non-zero, positive number of milliseconds for the first keyframe. As long as the elapsed time lies before the first keyframe, the `Interpolator` lies dormant. When the elapsed time falls after the last keyframe, the `Interpolator` stops interpolating the attribute. The attribute will remain at the value specified at the last keyframe until some other process modifies it.

TODO: cycling Interpolators!!

An `Interpolator` can only be a child of a `TransformGroup` and may not have any child elements.

2.1.5 Geometry

A `Geometry` element contains a list of `Point3` objects and a list of `Face`s. Each `Point3` has double-precision X,Y and Z components, while each `Face` describes a convex polygon as an array of indices into the `Point3` list.

On visiting a `Geometry` during scene graph traversal a `SceneRenderer` will first make a copy of it. The `SceneRenderer` then transforms the copy by each `TransformGroup` on the path up to the root before rendering it (see Section **??**). The `SceneRenderer` does not render `Face`s which are oriented away from the user (backfaces). Such faces are those for which the angle between the normal vector and the vector towards the viewpoint is greater than 90 degrees, where the normal is computed from the dot product of the first three vertices.

The only element that can be a child of a `Geometry` is an `Appearance` (described in Section **??**), to specify to a `SceneRenderer` the colours to apply to the edges and interiors of the `Geometry`'s `Face`'s when rendering or highlighting (see Section **??**).

2.1.6 Label

A `Label` element is a string of text with a `Font` and an offset on X,Y and Z axis specified by a `Point3`.

As with a Geometry (see Section ??), a SceneRenderer transforms a copy of each Label by each TransformGroup on the path from the Label up to the root before rendering the copy. Note that the Label's Font will diminish due to perspective projection.

The only element that can be a child of a Label is an Appearance (described in Section ??), to specify to a SceneRenderer the colours to apply when rendering or highlighting (see Section ??).

2.1.7 Appearance

An Appearance element specifies to a SceneRenderer colours to apply when rendering the Appearance's parent element. Four colours are defined by an Appearance:

- fillColor,
- edgeColor,
- highlightFillColor, and
- highlightEdgeColor.

For a Geometry parent, all four colours specified by Appearance are used to specify fill and edge colours for that parent's Faces (described in Section??), as well as alternative fill and edge colours for when highlighting (see Section ??).

For a Label parent, only the fillColour and highlightFillColour are employed to specify colours for normal rendering and highlighting of text.

An Appearance can only be a child of a Geometry or a Label, and may not have any child elements.

2.1.8 Name

A Name element defines a collective name for sub-elements.

The actual name value of a Name is held in it's Selector property of the Name. A Selector has a compare method which works just like java.lang.String's method of the same name, returning -1 if the first is of lower order than the second, 0 if they are equal, or 1 if the second is of lower order than the first.

As described in Section ??, if the mouse is clicked and released over a sub-element of a Name, the SceneRenderer will fire a MOUSE_PICKED event with a Selector identifying the Name.

To see how Name elements work, consider the example scene graph of Figure ??. If the mouse is clicked and released over any Face within the two Geometry elements, the SceneRenderer will fire a MOUSE_PICKED event with a Selector with the value "myHouse".

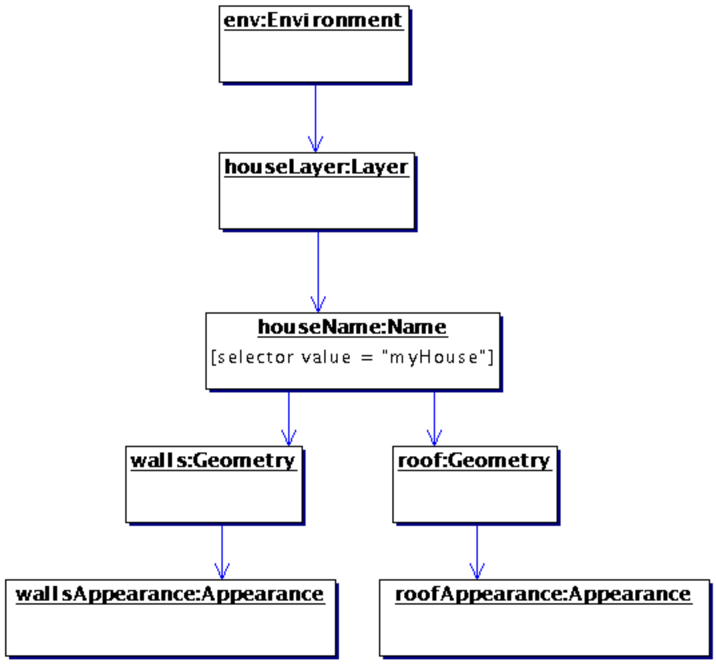

Figure 2: UML instance diagram of a scene graph with a `Name` element giving the collective name "myHouse" to `Geometry` elements which model the roof and walls of a house.

2.2 The Scene Graph Builder

Class `SceneBuilder` builds a scene graph that is guaranteed to be correctly formed. It is therefore the recommended method by which to build a scene graph from within Java.

The SceneBuilder works on a metaphor of nested opening and closing of scene elements such that the last element still open is the one to which commands apply. Any command issued that is not relevant to the open element is logged as an error and is otherwise ignored, allowing the build process to continue unaffected. This feature allows more information to be available at once when debugging a long list of building instructions. It can also provides a friendlier user experience when the SceneBuilder is directed by a file parser, where the parser does not neccessarily have to be restarted after each file correction in order to find the next error.

The following code snippet illustrates the use of a `SceneBuilder`, with indents to clarify nesting:

```
SceneBuilder builder = new SceneBuilder();
```

```
builder.openEnvironment();
    builder.addLightSource();
    builder.addLightSource();
    builder.openLayer();
        builder.openGeometry();
        builder.close();
    builder.close();
builder.close();
SceneElement scene = builder.buildScene();
```

3 The SceneRenderer

The `SceneRenderer` is shown in the class diagram of Figure ??. It is configured with a `SceneRendererParams` which specifies viewing parameters, and may be given a single `SceneRendererListener` to handle `SceneRendererEvents` fired during the rendering process. The following sections briefly describe these last three classes before describing in more detail the behaviour of a `SceneRenderer`.

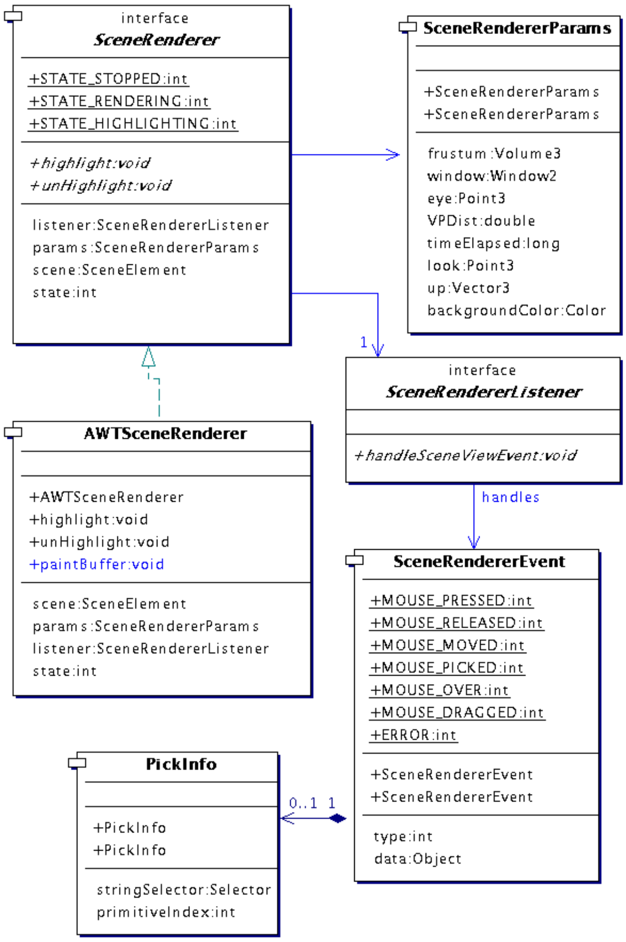

Figure 3: UML class diagram showing the abstract `SceneRenderer`, with implementation `AWTSceneRenderer` which adapts the renderer API to the `AWT Panel` class and provides double-buffering for flicker-free animation.

3.1 SceneRendererParams

Along with viewing parameters a `SceneRendererParams` also holds the elapsed scene time, which is updated by the `SceneRenderer` after each frame rendered. A `SceneRenderer` can be updated with a new `SceneRendererParams` at rendering time in order to do things such as move the viewpoint, move the projection plane distance for a zooming effect etc.

3.2 SceneRendererEvent

A `SceneRendererEvent` has a type identifier and an element of data, the type of which depends upon the type of the event. For `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_MOVED` and `MOUSE_DRAGGED` the data element will be a `Point2` indicating the location of the mouse.

For `MOUSE_PICKED` and `MOUSE_OVER` events the data will be a `PickInfo` identifying

- a `Selector` identifying the `Name` super-element of the nearest element on the Z-axis that intersects the mouse position and
- the index of the element's primitive (ie. `Face`) that was clicked on.

If the element picked/selected was a cube, for example, the primitive index would indicate which face (0-5) was clicked on.

3.3 SceneRenderer Behaviour

Figure **??** shows the three states of a `SceneRenderer`: `STATE_STOPPED`, `STATE_RENDERING` and `STATE_HIGHLIGHTING`. Initially a `SceneRenderer` is in `STATE_STOPPED`, where it has not yet been given a scene graph to render.

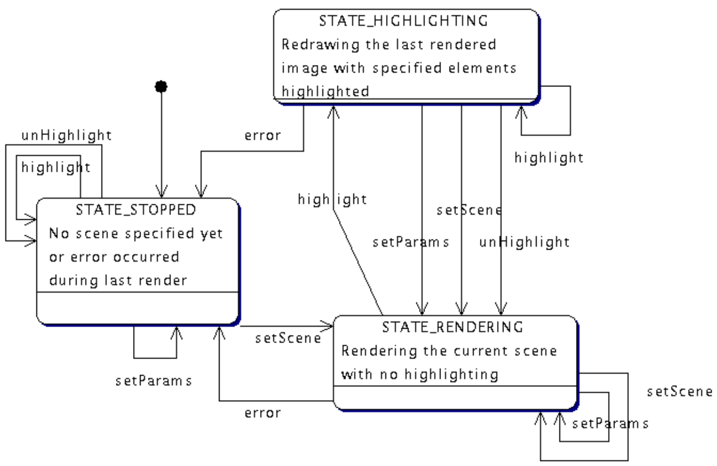

Figure 4: UML state diagram showing the states of a `SceneRenderer`

3.3.1 STATE_STOPPED

In this state the `SceneRenderer` awaits a scene graph to render. As soon as it gets one, it transitions to `STATE_RENDERING`. If `SceneRenderer` was not given a `SceneRenderParams` before it is given the scene graph, it will start `STATE_RENDERING` with it's own default `SceneRenderParams`.

Any directions to highlight or un-highlight are ignored in this state.

3.3.2 STATE_RENDERING

In this state the `SceneRenderer` continually renders it's scene, updating `Interpolator`s with the time elapsed for each frame.

If updated with new `SceneRenderParams` the `SceneRenderer` will continue `STATE_RENDERING`. A `SceneRenderer` stores the current elapsed time in it's `RenderParams`, so if the scene `Interpolators` are to continue where they left off before the update, the new `RenderParams` should have the same elapsed time value as the last `RenderParams`.

If updated with a new scene graph the renderer will continue `STATE_RENDERING` the new scene with time continuing to elapse from where it left off.

A highlight direction specifies a list of `Selector`s corresponding to scene `Name` elements. When directed to highlight when `STATE_RENDERING`, the `SceneRenderer` will transition to `STATE_HIGHLIGHTING`.

3.3.3 STATE_HIGHLIGHTING

In this state `SceneRenderer` starts by re-renderering it's last image with sub-elements of the specified `Name`s highlighted. The `SceneRenderer` will then remain `STATE_HIGHLIGHTING`, with time and `Interpolations` suspended.

If updated with new `SceneRenderParams` or a new scene the `SceneRenderer` will transition back to `STATE_RENDERING`.

If directed to highlight again, the `SceneRenderer` will again re-renderer it's last image, this time with subelements of the newly selected `Name` elements highlighted. As before, the `SceneRenderer` will remain `STATE_HIGHLIGHTING`.

Every time a `SceneRenderer` renders a scene in `STATE_RENDERING` it first generates an intermediate 2-D display list of the projections of those polygons which intersect the view volume, which it then renders to the display. Each time the `SceneRenderer` re-renders the last scene view in `STATE_HIGHLIGHTING` it only has to re-draw the 2-D display list, with polygons belonging to highlighted elements rendered in thier highlight colours.

3.3.4 Transitions on Error

Any error that occurs in `STATE_RENDERING` or `STATE_HIGHLIGHTING` states will cause the `SceneRenderer` to transition to `STATE_STOP`. Errors might be due to running out of memory, divide-by-zero or other conditions that cannot be predicted by scene graph validation.

4 Example of Use

In this example we will set up a renderer, build a scene graph, then render the scene graph with the renderer. The scene graph we will build is shown in the instance diagram of Figure ??.

4.1 Setting up the SceneRenderer

Let's set up an AWT-compatible `AWTSceneRenderer` then attach a `SceneListener` to handle `SceneRendererEvents`. This example only handles `MOUSE_PICKED` events.

```
...

SceneRenderer renderer = new AWTSceneRenderer();

renderer.setListener(
    new SceneListener() {
        public void handleEvent(SceneRendererEvent e) {
            switch (e.getType()) {
                case SceneRendererEvent.ERROR:
                    System.out.println((String)e.getData());
                    break;

                case SceneRendererEvent.MOUSE_DRAGGED:
                    break;

                case SceneRendererEvent.MOUSE_MOVED:
                    break;

                case SceneRendererEvent.MOUSE_OVER:
                    break;

                case SceneRendererEvent.MOUSE_PRESSED:
                    break;

                case SceneRendererEvent.MOUSE_RELEASED:
                    break;

                case SceneRendererEvent.MOUSE_PICKED:
                    PickInfo pickInfo = (PickInfo)e.getData();
                    Selector name = pickInfo.getSelector();
                    System.out.println("Pick event handled: \""
                        + name.toString() + "\"");
                    break;
                default:
        }
    }
});

SceneRendererParams params = new SceneRendererParams();
```

```
params.setFrustum(new Volume3(-200.0, -200.0, -300.0, 200.0, 200.0, -100.0));
params.setWindow(new Window2(0, 0, 800, 800));
params.setVPDist(-500.0);
params.setEye(new Point3(0.0, 0.0, 100.0));
params.setLook(new Point3(0.0, 0.0, 0.0));
params.setUp(new vector3(0.0, 1.0, 0.0));
params.setBackgroundColor(new Color(100, 255, 255)); // Light blue

renderer.setParams(params);

// At this point we would add the renderer to a layout

...
```

Our renderer is now in the STATE_STOPPED state and awaits a scene graph.

4.2 Building the Scene Graph

Let's create the scene graph (see Figure **??**) using a SceneBuilder. We'll attach an ErrorHandler to the SceneBuilder to report any errors that occur during the build process, then we'll issue commands to the SceneBuilder to create a scene containing a red cube with yellow edges that rotates 360 degrees over five seconds before stopping. The cube will be illuminated by two light sources, one yellow, the other white. Although probably not noticeable in this example, the cube will fade into a light blue background in proportion to it's depth in the view space (depth cueing). The cube also gets a name, "Hello, World", which our renderer will print to standard output when the cube is clicked on (picked).

```
SceneBuilder builder = new SceneBuilder();

builder.setErrorHandler(new ErrorHandler() {
    public void handleError(String message) {
        System.out.println("Error building scene: " + message);
    }
});

builder.openEnvironment();
builder.addLightSource(new Vector3(1.0,  0.0, 0.0), new Color(60,60,0));
builder.addLightSource(new Vector3(1.0, -1.0, 0.0), new Color(100,100,100));
builder.openLayer("foreground", true, true, true); // Depthsort, depth-cueing and shading
builder.openTransformGroup();
builder.rotateY(0.0);
builder.rotateX(45.0);
builder.openName(new StringSelector("Hello, World!"));
builder.openBox(30.0, 30.0, 30.0);
builder.openMaterial();
builder.setEdgeColor(new Color(255,255,0));        // Yellow
builder.setFillColor(new Color(255,0,0));          // Red
builder.close();
```

```
builder.close();
builder.close();
builder.openInterpolator(TransformGroup.ROTY_VAL);
builder.addKeyFrame(0, 0.0);
builder.addKeyFrame(5000, 360.0);                    // Five seconds
builder.close();                                     // Interpolator
builder.close();                                     // TransformGroup
builder.close();                                     // Layer
builder.close();                                     // Environment

SceneElement  sceneRoot = builder.buildScene();
```

All that remains to do now is hand the scene to the renderer, which then immediately begins rendering the scene:

```
renderer.setScene(sceneRoot);                        // Rendering starts now
```

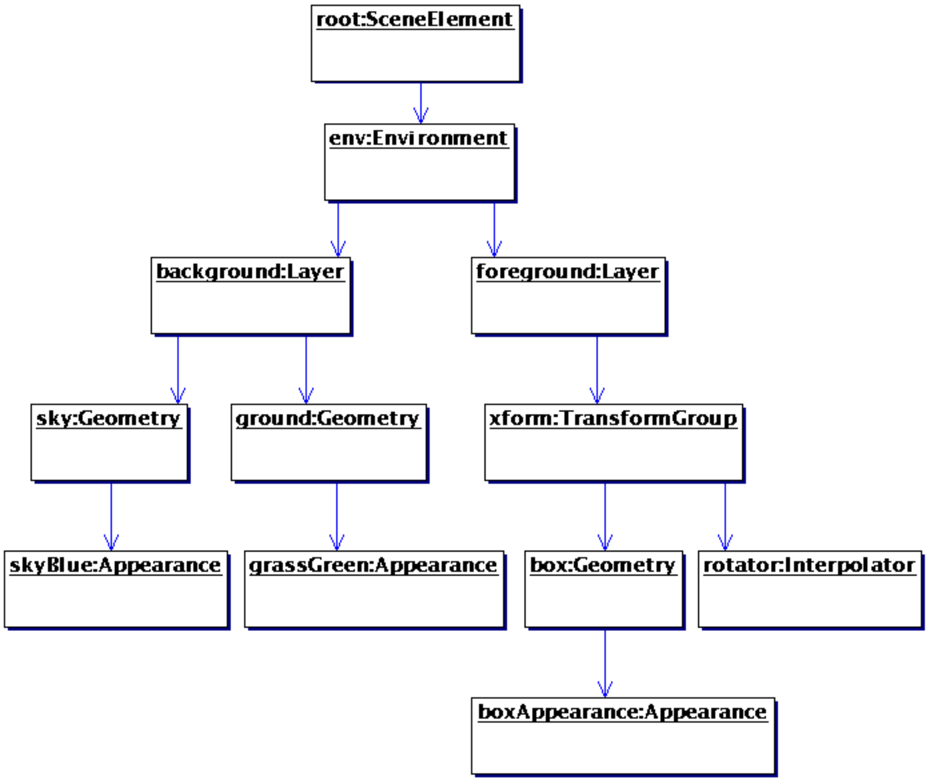

Figure 5: An example scene graph containing a rotating box against backdrop of blue sky and green grass.