4 COLLADA Scenes

Overview

This chapter discusses COLLADA scene composition and object instantiation, including the concepts of materials, lights, cameras, and geometry within the visual scene.

Materials

Every computer graphics system has a concept of rendering and a methodology on how to represent that information in the object to be drawn or rendered on the display device. In the real world, we also have a notion of an object's visual appearance or what it looks like to us. When asked what a floor looks like, we answer, "It looks like concrete" or "It looks like redwood." Our response is to describe the material of the floor. This is a fairly abstract answer that relies on the questioner's own knowledge of "concrete" or "redwood" to visualize the floor's appearance. By identifying the material of the floor, we have given a high-level description of its appearance.

The COLLADA design also uses the concept of material to represent the appearance of geometry at the highest level [43]. The `<material>` element is the container for all the information that describes the visual appearance of a geometric object (see Figure 4.1). It is composed of three elements:

- an optional `<asset>` element that contains the asset metadata;
- an `<instance_effect>` element that declares the rendering effect that achieves the material's visual appearance;
- an optional `<extra>` element that contains additional user-defined or application-specific information.

COLLADA design relies entirely on COLLADA FX and the `<effect>` element to describe the rendering state for a material. The `<instance_effect>` element instantiates the effect within the `<material>` element.

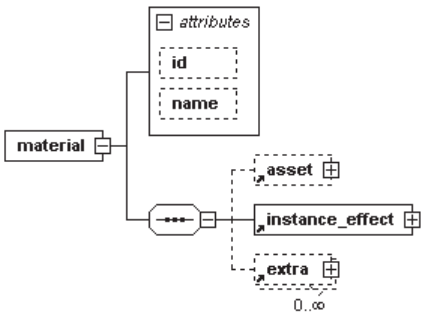


Figure 4.1. The <material> element.

As explained in the previous chapter, geometric primitives, such as the <polygons> element, have a material attribute that declares a symbolic name for a <material> element that will ultimately be bound to it during the instantiation process. The name attribute defines the material's symbolic name for this purpose.

```
<polygons material="Blue" count="6">
...
<material name="Blue" id="Unique_Blue">
```

Materials are assets and therefore have an id attribute so they can be referenced.

Materials are instances of effects. Effects are shading algorithms that are written in a variety of shading languages, such as Cg or GLSL, and packaged with a variety of programmable state and other platform-specific configuration settings. Effects are executed on modern graphics processing units (GPUs) as a series of shader programs. To learn more about effects and shaders, please refer to Chapter 5, "COLLADA Effects," page 91.

When instantiating an effect, any parameter of the effect can be changed by the material. For example, a "shiny_effect" can be used to create a "blue_shiny" material and a "gold_shiny" material by adjusting the appropriate color parameter in the <material> element.

```
<effect id="shiny_effect">
 <profile_COMMON>
  <technique>
   <newparam sid="Diffuse_Color">
    <semantic>DIFFUSE</semantic>
    <float3> 0.8 0.8 0.8 </float3>
   </newparam>
   <newparam sid="Super_shiny">
```

```
    <semantic>SHININESS</semantic>
    <float> 0.2 </float>
   </newparam>
   <phong/>
  </technique>
 </profile_COMMON>
</effect>

<material name="blue_shiny">
 <instance_effect url="#shiny_effect">
  <setparam ref="Diffuse_Color">
   <float3> 0.0 0.0 1.0 </float3>
  </setparam>
 </instance_effect>
</material

<material name="gold_shiny">
 <instance_effect url="#shiny_effect">
  <setparam ref="Diffuse_Color">
   <float3> 0.8 0.6 0.2 </float3>
  </setparam>
 </instance_effect>
</material>
```

The previous example shows a single `<effect>`, identified as `"shiny_effect"`, which defines two new parameters. One of those parameters has a `sid` value of `"Diffuse_Color"`. The `"blue_shiny"` material sets that parameter, using the `<setparam>` element, to a pure blue (RGB) color value. The `"gold_shiny"` material sets it to a gold color value. These settings take place when the material is bound to some geometry in the scene.

To encourage document modularity, materials are declared and defined inside `<library_materials>` elements. A COLLADA document that contains only a set of materials in a library can define a visual theme, such as `"hardwood_floors"`, that is reused by many different documents that share the floor descriptions with each other, for example:

```
<COLLADA>
 <asset/>
 <library_materials id="hardwood_floors">
  <asset/>
  <material name="walnut" />
  <material name="oak" />
  <material name="mahogany" />
 </library_materials>
</COLLADA>
```

Lights

Lights are absolutely necessary for computing the visual representation of an object. Without a light source, everything in the scene will be rendered black on black for most geometry. Some lighting models include the concept of glowing or emissive geometry that do not require a light source to produce their color. Lights are not part of the geometry or material definition. Lights are independently defined and are important parameters to the rendering algorithm along with the given geometry and material.

The number of active lights is often limited in the graphics hardware or software due to the high cost associated with each additional light in the calculations. Even so, the quality of the rendering is directly proportional to the number of light sources in the scene, so artists are constantly striving to make the best use of all available lights. This is a major reason why lights are a distinct element in the design of COLLADA even though global illumination models enable any and all objects in the scene to emit light and contribute to the final rendering result [44].

To be active, a light must be instantiated into the visual scene. The directional, point, and spot lights take their position and orientation from the coordinate system of the parent `<node>` element in the scene (see Figure 4.2). The ambient light is not affected by such spatial transformations since it radiates omnidirectionally from everywhere in the scene equally.

The basic light source representation is defined by the light's `<technique_common>` child element (see Figure 4.3). It can contain one of four possible choices of light types that are common among graphics applications and fixed-function rendering implementations. The four choices are ambient, directional, point, and spot [45]. Each type of light has specific properties that artists use to create the visual lighting effects. The simpler ambient light requires fewer calculations than the complex spot light.

Ambient Light

Ambient light radiates energy from every direction equally. Its light appears to come from everywhere at once. The intensity of the light is not attenuated by distance or viewing angle. This is the simplest type of light and the cheapest way to add color to objects in the scene.

```
<xs:element name="ambient">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="color" type="TargetableFloat3"/>
  </xs:sequence>
```

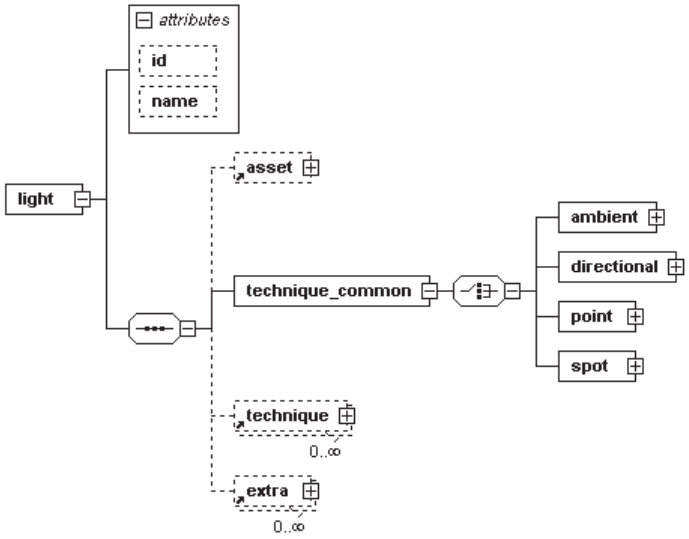


Figure 4.2. The <light> element.

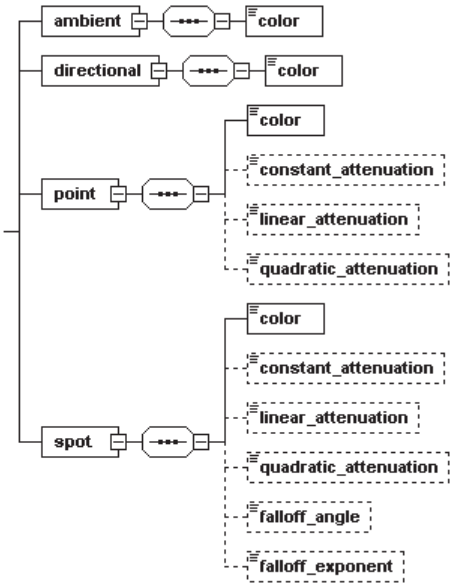


Figure 4.3. The <light><technique_common> element.

```
 </xs:complexType>
</xs:element>
```

As you can see from the schema for the <ambient> element, it defines a single <color> element that contains the RGB color value for the light. The ambient color value can be targeted for animation as discussed in Chapter 6, "COLLADA Animations," page 121.

Directional Light

Directional light radiates energy from one direction. The light source is always infinitely far away so the light rays are parallel to the direction everywhere in the scene. The intensity of the light is not attenuated. This type of light is often used to model the sun, moon, and stars.

```
<xs:element name="directional">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="color" type="TargetableFloat3"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The <directional> element is also relatively simple, defining only a <color> element. Its type implicitly means that it has a direction vector that shines the light down its local negative *Z*-axis. This direction can be reoriented by rotating the light when it is placed in the scene.

Point Light

Point light radiates energy from a single location in all directions equally. The intensity of this light can be attenuated by its distance from the viewer. This light is probably the most common light because it models most everyday light sources, such as candles and lamps.

```
<xs:element name="point">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="color" type="TargetableFloat3"/>
   <xs:element name="constant_attenuation"
         type="TargetableFloat"
         default="1.0" minOccurs="0"/>
   <xs:element name="linear_attenuation"
         type="TargetableFloat"
         default="0.0" minOccurs="0"/>
   <xs:element name="quadratic_attenuation"
         type="TargetableFloat"
```

```
          default="0.0" minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The <point> element is noticeably more complicated than either the <ambient> or <directional> elements. Not only does it define a <color> element, but also three types of attenuation: constant, linear, and quadratic. Each attenuation element is optional and has a default value if it is not supplied. These elements define values for three terms in the attenuation formula:

$$\text{Attenuation} = A1 + A2 + A3$$
$$\text{where} \quad A1 = \mathit{ConstAttenuation}$$
$$A2 = (\mathit{Dist} \times \mathit{LinearAttenuation})$$
$$A3 = (\mathit{Dist}^2 \times \mathit{QuadraticAttenuation})$$

The location of the point light is established when it is placed in the scene. For example, here we place a point light at the local coordinates X=50, Y=30, and Z=20 using a <translate> element.

```
<library_lights>
 <light id="lamp">
  <technique_common>
   <point>
    <color> 1.0 1.0 0.2 </color>
   </point>
  </technique_common>
 <light>
</library_lights>

<node>
 <translate> 50.0 30.0 20.0 </translate>
 <instance_light url="#lamp"/>
</node>
```

Spot Light

The spot light is similar to a point light that shines a beam of light along the direction it is facing. The beam expands wider, in a cone shape, as the distance from the light increases. This type of light source is used to model flashlights, searchlights, headlights, and, of course, spotlights.

A spot light's brightness is attenuated by its distance from the viewer, by its orientation, and also by the angle of the beam away from the light's direction. The attenuation due to distance uses the same formula as for point lights.

```
<xs:element name="spot">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="color" type="TargetableFloat3"/>
   <xs:element name="constant_attenuation"
         type="TargetableFloat"
         default="1.0" minOccurs="0"/>
   <xs:element name="linear_attenuation"
         type="TargetableFloat"
         default="0.0" minOccurs="0"/>
   <xs:element name="quadratic_attenuation"
         type="TargetableFloat"
         default="0.0" minOccurs="0"/>
   <xs:element name="falloff_angle"
         type="TargetableFloat"
         default="180.0" minOccurs="0"/>
   <xs:element name="falloff_exponent"
         type="TargetableFloat"
         default="0.0" minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The <spot> element has an implied direction facing down the negative *Z*-axis, just like the <directional> element. The actual orientation is established when the light is placed in the scene, as with the other light types.

The shape of the spot light beam is defined by the <falloff_angle> and <falloff_exponent> child elements. The value of <falloff_angle> can range from 0 to 90 degrees, or it can have the special value of 180 degrees for a uniform distribution, just like a point light source. The value of <falloff_exponent> can range from 0 to 128. Higher exponent values focus the beam, making it brighter in the center and dimmer at the perimeter of the beam, as a function of the cosine of the angle between the light direction and the surface (normal) of the object it is shining on.

Camera

The <camera> element defines viewing parameters (see Figure 4.4). Like a real camera, it has optical and imaging properties that can be adjusted to suit the needs of the user. To change the field of view on a camera, you change the lens, assuming it's not a fixed-lens or disposable camera. More often, you change the film to match the lighting conditions, using a slow exposure film on a bright sunny day. Although analog film is on the decline with the advent of digital cameras, the comparison still holds. In COLLADA, the <optics>

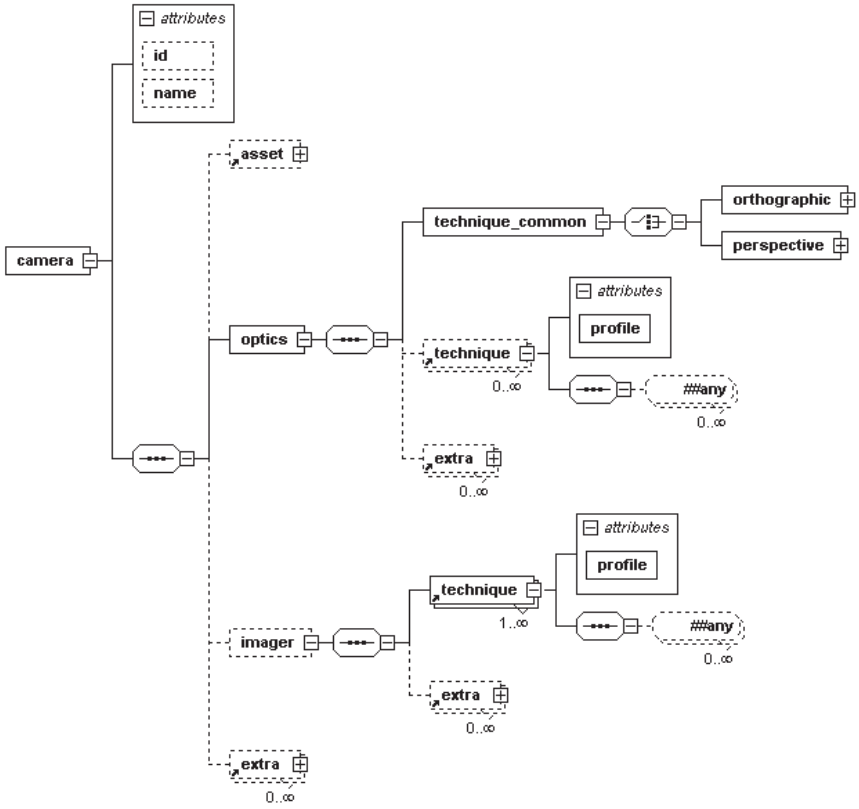


Figure 4.4. `<camera>` element.

element models the lens of the camera. The `<imager>` element models the film loaded into the camera.

A camera's position and orientation are established when it is placed in the scene. By convention, a camera's implicit orientation has it looking down its *Z*-axis, with its *X*-axis to the left and its *Y*-axis upward. As with other objects that can be placed in the scene, cameras are defined within a library module using the `<library_cameras>` element.

```
<library_cameras>

 <camera id="my_camera2">
  <asset/>
  <optics>
   <technique_common>
    <orthographic>
```

```
     <xmag>1000.0</xmag>
     <ymag>700.0</ymag>
     <znear>0.1</znear>
     <zfar>10000.0</zfar>
    </orthographic>
   </technique_common>
   <extra/>
  </optics>
  <imager>
   <technique/>
   <extra/>
  </imager>
  <extra/>
 </camera>
....

 <camera id="my_camera1">
  <asset/>
  <optics>
   <technique_common>
    <perspective>
     <xfov>1000.0</xfov>
     <yfov>500.0</yfov>
     <znear>2.0</znear>
     <zfar sid="animated_far">100.0</zfar>
    </perspective>
   </technique_common>
   <extra/>
  </optics>
  <extra/>
 </camera>

</library_cameras>

<node>
 <translate> 100.0 0.0 -100.0 </translate>
 <instance_camera url="#my_camera2"/>
</node>
```

Optics

The <optics> element models the camera lens and is extensible. The properties most often needed are defined within the <technique_common> element. Alternatively, artists can supply their own definition using the <technique> element. In either case, additional properties can be given using the <extra> element.

Within the common technique, two types of lens are provided by the <orthographic> and <perspective> elements. The orthographic lens projects the scene onto the plane of the film (<imager>) so that the lines of projection are parallel to the plane. The resulting image lacks any depth perspective. Conversely, the perspective lens preserves the eye's natural perception of objects using a projection where each line extends to its own vanishing point. The eye is able to discern three-dimensional perception of the resulting image and to retain a sense of geometric perspective, hence the name of this projection.

The schema for these two elements look more complicated than they really are, as they offer some choices in how the parameters are specified. Using the <orthographic> element's schema, we will explain the choices. First, the schema is as follows:

```
<xs:element name="orthographic">
 <xs:complexType>
  <xs:sequence>
   <xs:choice>
    <xs:sequence>
     <xs:element name="xmag" type="TargetableFloat"/>
     <xs:choice minOccurs="0">
      <xs:element name="ymag" type="TargetableFloat"/>
      <xs:element name="aspect_ratio"
            type="TargetableFloat"/>
     </xs:choice>
    </xs:sequence>
    <xs:sequence>
     <xs:element name="ymag" type="TargetableFloat"/>
     <xs:element name="aspect_ratio"
           type="TargetableFloat" minOccurs="0"/>
    </xs:sequence>
   </xs:choice>
   <xs:element name="znear" type="TargetableFloat"/>
   <xs:element name="zfar" type="TargetableFloat"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

To specify an orthographic projection, we need to know the X- or Y-axis magnification at a minimum (in conjunction with the display devices viewport size). We also want the option of providing both or either magnifications and the aspect ratio. Finally, we need to know the near and far clipping planes to complete the viewport viewing frustum.

```
<orthographic>
 <xmag> 1280 </xmag>
 <ymag> 1024 </ymag>
</orthographic>

<perspective>
 <yfov> 50.0 </yfov>
 <aspect_ratio> 0.88 </aspect_ratio>
</perspective>
```

The schema for these optical elements allows the artist to use these choices without needing to supply more information than is required. For example, it would be a validation error to supply all three elements of <xmag>, <ymag>, and <aspect_ratio> at once.

```
<orthographic>
 <xmag> 1280 </xmag>
 <ymag> 1024 </ymag>
 <aspect_ratio> 1.25 </aspect_ratio> <!-- validation error -->
</orthographic>
```

Imager

The <imager> element is like camera film and is fully extensible [46]. There are no common properties for this element because existing DCC tools do not have a consistent methodology for this aspect of rendering. The COLLADA design group considered using ASA/ISO film speed ratings, but the decision was made to leave it open to extension using the <technique> and <extra> elements.

Node

So far, this chapter has discussed placing items in the scene, such as the lights and cameras. The first step in this process is to create a node hierarchy of coordinate transformations. Each node in the hierarchy defines its own local coordinate system (see Figure 4.5). Any object contained in the node inherits these coordinates and therefore its placement. This is the primary job of the <node> element [47].

The node is an asset and can be extended with additional information in the <extra> element. The <node> element has several optional attributes, among them

- id and name attributes for identification;
- sid attribute for animation targeting;
- layer and type attributes as an aid to rendering and scene-graph software.

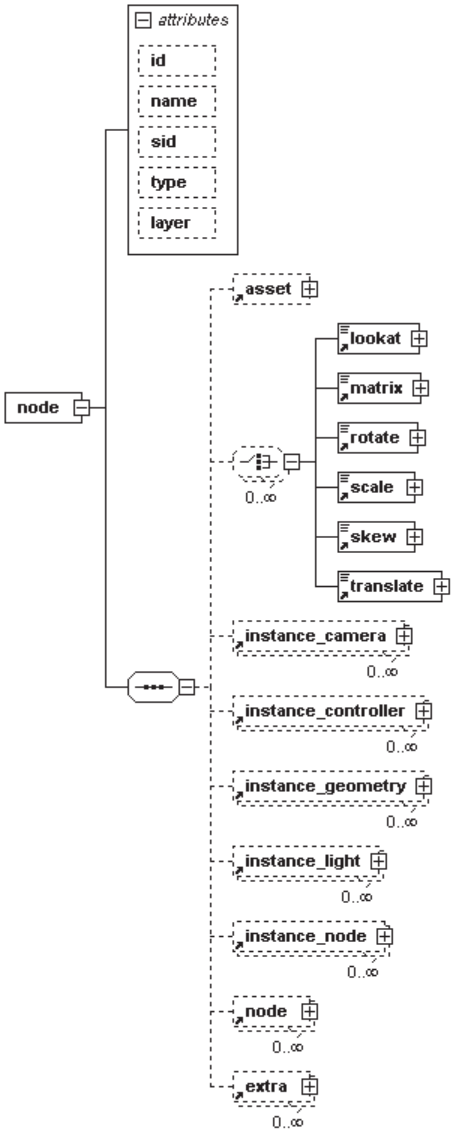


Figure 4.5. The <node> element.

The `type` attribute is useful when creating articulated characters and similar models. A `type` value of "`JOINT`" distinguishes a skeleton joint from other nodes in the hierarchy.

A <node> element has the following optional attributes:

Attribute	Type	Default
id	xs:ID	—
layer	ListOfNames	—
name	xs:anyURI	—
sid	xs:NCName	—
type	NodeType	"NODE"

The layer attribute is somewhat unique in that it contains a list of names and not a single value. This is because the node can belong to several rendering layers at once, and so more than one layer name can be supplied, separated by whitespace. For example, the following node has membership in three layers:

```
<node id="hill_0013" layer="ground mountains California">
 <instance geometry url="#some_hill"/>
</node>
```

The <node> element represents the transformation hierarchy recursively, so it can have itself as a child element. The collection of nodes form a directed acyclic graph (DAG) or tree. It has the same structure as the scene graph that is often used in higher-level rendering software.

```
<node name="Earth">
  <node name="Europe">
    <node name="France">
      <node name="Paris"/>
      <node name="Grenoble"/>
    </node>
    <node name="UnitedKingdom">
      <node name="Wales"/>
      <node name="Scotland"/>
    </node>
  </node>
</node>
```

The relationship of the parent nodes to their child nodes is arranged spatially so that objects sharing the same coordinate system will have the same parent node. Traversing from the root of the tree to the object, node by node, enables software to process and render the scene efficiently. To define its own local coordinate system, each <node> element can have any number of three-dimensional transformation elements to translate, rotate, and scale it. The set of possible transformation elements are in the following list:

- <lookat> element represents a position and orientation;
- <matrix> element represents an accumulation of transformations;
- <rotate> element represents a rotation about an axis;
- <scale> element represents a nonuniform scale;
- <skew> element represents a shearing translation and rotation;
- <translate> element represents a translation.

These transformations can be combined in any number and ordering to produce the desired coordinate system for the parent <node> element. The COLLADA specification requires that the transformation elements are processed in order and accumulate their result as if they were converted to column-order matrices and concatenated using matrix post-multiplication. Therefore, import and export software must preserve the ordering at all times; otherwise, the coordinate systems will not remain correct between applications. To illustrate, here are some transformations injected into the previous node tree example.

```
<node name="Earth">
 <translate sid="T1"> 500.0 123456.0 0.0 </translate>
 <rotate sid="R1"> 36.0 0.0 1.0 0.0 </rotate>
 <node name="Europe">
  <matrix sid="M1"> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  </matrix>
  <node name="France">
   <lookat sid="L1"> 23000 10000 400 0 0 0 0 1 0 </lookat>
  </node>
  <node name="UnitedKingdom">
   <translate sid="T2"> -254 965 -22 </translate>
  </node>
 </node>
</node>
```

With this example, we have given each node the following coordinate systems:

- Earth = (T1) · (R1);
- Europe = (T1) · (R1) · (M1);
- France = (T1) · (R1) · (M1) · (L1);
- UnitedKingdom = (T1) · (R1) · (M1) · (T2).

Within the context of each <node> element, there is a well-defined coordinate system in which we can instantiate (create instances of) geometry and

other objects. A family of elements is designed to instantiate a specific type of object, as follows:

- <instance_camera> element creates an instance of a <camera>;
- <instance_controller> element creates an instance of a <controller>;
- <instance_geometry> element creates an instance of a <geometry>;
- <instance_light> element creates an instance of a <light>;
- <instance_node> element creates an instance of a <node>.

All of these instancing elements perform a similar operation. They take an object defined in a corresponding library element and create a copy of it in the coordinate system of the node. In this manner, objects are placed into a spatially organized tree of information that describes the visual scene.

Visual Scene

The <visual_scene> element contains all the information that describes the visual aspects of the scene and often some related data that is tightly coupled to it by authoring tools (see Figure 4.6). The main example is animation data that is tightly coupled to the visual geometry of an articulated character.

In the original design of COLLADA, the visual scene was the only type of scene. Later when another domain was introduced into COLLADA (namely, physics), the concept of multiple domains was formalized. The name of the element was changed, to <visual_scene>, with the expectation that more

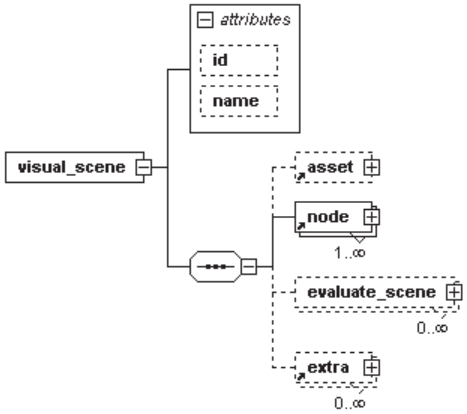


Figure 4.6. The <visual_scene> element.

types of scenes would be introduced as COLLADA expands to cover more domains of interest [48].

The visual scene is an asset and has an optional `id` attribute so it can be referenced. It can also have a `name` attribute and can be extended directly using `<extra>` elements.

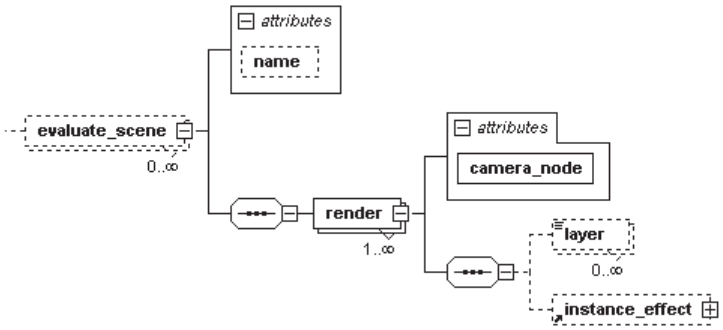
Each `<visual_scene>` element must contain at least one `<node>` element. The nodes contain the geometry, lights, and cameras that make up the scene. The `<visual_scene>` element acts as the root of the node hierarchy (scene graph).

Next comes an optional `<evaluate_scene>` element (see Figure 4.7). There can be more than one per visual scene.

The `<evaluate_scene>` element acts as a simple form of command scripting [49] as represented by a sequence of `<render>` child elements. It contains the information needed to process the scene contents for each rendering pass. This information includes layer designations, effects instantiation, and the choice of camera to be used as the observer. The details of effects are discussed in the next chapter. The layer names are matched against the `layer` attribute values of the `<node>` elements to select only the matching nodes during the current evaluation pass. One camera is needed during the evaluation, so the `camera_node` attribute is required.

Scene

Each COLLADA document can contain one scene. This scene represents the instantiation of visual objects and physical simulations that comprise the



Attribute	Type	Default
name	xs:NCName	—
camera_node	xs:anyURI	required

Figure 4.7. The `<evaluate_scene>` element.

content created by the artist. The <scene> element may embody a single character, an inanimate rock, or a detailed cityscape used as a level in a video game.

It may seem like a good idea to be able to store more then one scene per document, but there are several reasons why the document and the scene have a one-to-one relationship. The document is the unit of storage and transmission of XML-encoded content, just as the game level is for video games. Loading in a game level is therefore a simple matter of loading the document that represents it. The one-to-one relationship also makes it easy to discern what content within the document is actually in the scene. Documents can be complex, containing large amounts of information. By easily finding the <scene> element and parsing the information backward from it, software tools can efficiently load the content that is contained only in that scene.

The <scene> element can instantiate one visual scene and any number of physical simulations (see Figure 4.8). It can be extended with the <extra> element as well. The one-to-one relationship between the scene and its visual representation is designed to maintain some simplicity that might otherwise become an explosion of content if it were within a single document. It is possible that this constraint will be relaxed in a future revision of the COLLADA schema.

```
<scene>
 <instance_physical_scene url="#earthly_physics"/>
 <instance_visual_scene url="#earthly_visuals"/>
 <extra type="Weather"/>
</scene>
```

As with the <node> element described previously, the <scene> element has elements that can create instances of specific types, as follows:

- <instance_physics_scene> element creates an instance of a physics simulation;

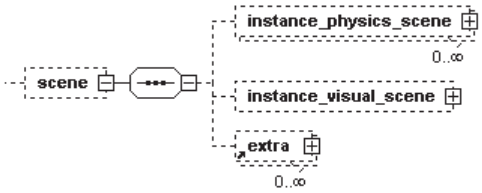


Figure 4.8. The <scene> element does not have any attributes.

- <instance_visual_scene> element creates an instance of a visual scene.

The actual content for these instantiations is defined in a library element of the corresponding type, for example:

```
<library_physics_scenes>
  <physics_scene id="earthly_physics">
  </physics_scene>
</library_physics_scenes>

<library_visual_scenes>
  <visual_scene id="earthly_visuals">
  </visual_scene>
</library_visual_scenes>
```

The <physics_scene> element is described in the section "Physics Scenes" in Chapter 7.

Example Scene

Bringing it all together into a comprehensive example that includes materials, geometry, lights, cameras, and action! (… I mean animation) will fill up several pages with XML. Instead, here are the salient points we have covered in this chapter in an outline-style example.

First, we have the libraries of information organized by type. The simple object declarations for this outline include a camera, geometry, an effect, and a light.

```
<library_cameras>
 <camera id="Eyes"/>
</library_cameras>

<library_geometries>
 <geometry id="Globe"/>
</library_geometries>

<library_effects>
 <effect id="Shiny"/>
</library_effects>

<library_lights>
 <light id="Sun_Light"/>
</library_lights>
```

The material is a little bit more complicated since it references the effect.

```
<library_materials>
 <material id="Shiny_Blue">
```

```
  <instance_effect url="#Shiny"/>
 </material>
</library_materials>
```

The node binds several things together including the geometry, material, and the light.

```
<library_nodes>
 <node id="Earth">
  <instance_geometry url="#Globe">
   <bind_material>
    <technique_common>
     <instance_material url="Shiny_Blue"/>
    </technique_common>
   </bind_material>
  </instance_geometry>
 </node>
 <node id="Sun">
  <instance_light url="#Sun_Light"/>
 </node>
</library_nodes>
```

Next, the visual scene creates the scene graph from the available nodes. A camera can be created here or be part of the incoming node hierarchy already. In this example, we'll create the camera here.

```
<library_visual_scenes>
 <visual_scene id="World">
  <node id="Root">
   <instance_node url="#Earth"/>
   <node>
    <matrix sid="Roving"/>
    <instance_camera url="#Eyes"/>
   </node>
  </node>
 </visual_scene>
</library_visual_scenes>
```

Finally, the visual scene is instantiated in the document.

```
<scene>
 <instance_visual_scene url="#World"/>
</scene>
```