Przetwarzanie i renderowanie modeli w formacie VRML

Michalis Kamburelis

Praca magisterska pod kierunkiem
dr Andrzeja Łukaszewskiego

Instytut Informatyki
Uniwersytet Wrocławski

Wrocław, Wrzesień 2006

VRML processing and rendering engine

Michalis Kamburelis

VRML processing and rendering engine

Michalis Kamburelis

Table of Contents

List of Figures

Goals

This thesis describes the implementation of a 3D engine based on the VRML language.

The *VRML language* is used to define 3D environments. It will be described in detail in Chapter 1, *Overview of VRML*. It has many advantages over other 3D languages:

- The specification of the language is open.
- The language is implementation-neutral, which means that it's not “tied” to any particular rendering method or library. It's suitable for real-time rendering (e.g. using OpenGL or DirectX), it's also suitable for various software methods like ray-tracing. This neutrality includes the material and lighting model described in VRML 2.0 specification.

  Inventor, an ancestor of the VRML, lacked such neutrality. Inventor was closely tied to the OpenGL rendering methods, including the OpenGL lighting model.
- The language is quite popular and many 3D authoring programs can import and export models in this format. Author of this document can recommend the open-source Blender [http://www.blender3d.org/] modeler.
- The language can describe geometry of 3D objects with all typical properties like materials, textures and normal vectors.
- The language is not limited to 3D objects. Other important environment properties, like lights, the sky, the fog, viewpoints, collision properties and many other can be expressed.
- The language is easy to extend. You can easily add your own nodes and fields (and I did, see the list of my VRML extensions [http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_extensions.php]).

Implementation goals were to make an engine that

- Uses VRML. Some other 3D file formats are also supported (in particular the popular 3DS format) by silently converting them to VRML.
- Allows to make a general-purpose VRML browser. See view3dscene [http://www.camelot.homedns.org/~michalis/view3dscene.php].
- Allows to make more specialized programs, that use the engine and VRML models as part of their job. For example, a game can use VRML models for various parts of the world:
  - Static environment parts (like the ground and the sky) can be stored and rendered as one VRML model.
  - Each creature, each item, each “dynamic” object of the world (door that can open, building that can explode etc.) can be stored and rendered as a separate VRML model.

  When rendering, all these VRML objects can be rendered within the same frame, so that user sees the complete world with all objects.

  Example game that uses my engine this way is “The Castle” [http:// www. camelot. homedns.org/~michalis/castle.php].

- Using the engine should be as easy as possible, but at the same time OpenGL rendering must be as fast as possible. This means that a programmer gets some control over how the engine will optimize given VRML model (or part of it). Different world parts may require entirely different optimization methods:
  - static parts of the scene,
  - parts of the scene that move (or rotate or scale etc.) only relatively to the static parts,
  - parts of the scene that frequently change inside (e.g. a texture changes or creature's arm rotates).

  All details about optimization and animation methods will be given in later chapters (see Chapter 5, *OpenGL rendering* and Chapter 6, *Animation*).

- The primary focus of the engine was always on 3D games, but, as described above, VRML models can be used and combined in various ways. This makes the engine suitable for various 3D simulation programs (oh, and various game types).

- The engine is open-source (licensed on GNU General Public License).

- Developed in object-oriented language. For me, the language of choice is ObjectPascal, as implemented in the Free Pascal compiler [http://www.freepascal.org].

Chapter 1. Overview of VRML

This chapter is an overview of VRML concepts. It describes the language from the point of view of VRML author. It teaches how a simple VRML files look like and what are basic building blocks of every VRML file. It's intended to be a simple tutorial into VRML, not a complete documentation how to write VRML files. If you want to learn how to write non-trivial VRML files you should consult VRML specifications.

This chapter also describes main differences between VRML 1.0 and 2.0. Our engine handles both VRML versions, and in the future X3D will be supported too. X3D is the successor of VRML 2.0 format (you can think of it as "VRML 3.0").

1.1. First example

VRML files are normal text files, so they can be viewed and edited in any text editor. Here's a very simple VRML 1.0 file that defines a sphere:

```
#VRML V1.0 ascii

Sphere { }
```

The first line is a header. It's purpose is to identify VRML version and encoding used. Oversimplifying things a little, every VRML 1.0 file will start with the exact same line: `#VRML V1.0 ascii`.

After the header comes the actual content. Like many programming languages, VRML language is a free-form language, so the amount of whitespace in the file doesn't really matter. In the example file above we see a declaration of a *node* called `Sphere`. “Nodes” are the building blocks of VRML: every VRML file specifies a directed graph of nodes. After specifying the node name (`Sphere`), we always put an opening brace (character `{`), then we put a list of *fields* and *children nodes* of our node, and we end the node by a closing brace (character `}`). In our simple example above, the `Sphere` node has no fields specified and no children nodes.

The geometry defined by this VRML file is a sphere centered at the origin of coordinate system (i.e. point (0, 0, 0)) with a radius 1.0.

1. Why the sphere is centered at the origin ?

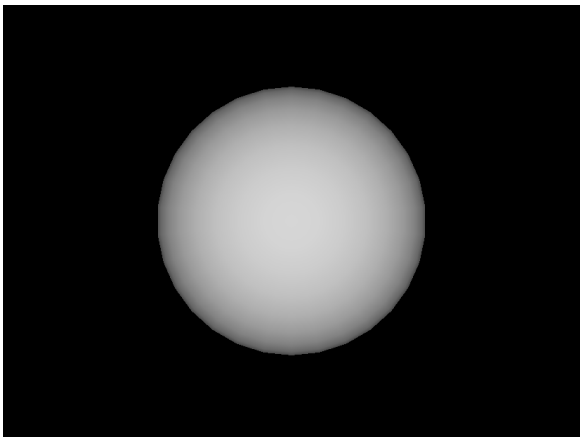
   Spheres produces by a `Sphere` node are always centered at the origin — that's defined by VRML specifications. Don't worry, we *can* define spheres centered at any point, but to do this we have to use other nodes that will move our `Sphere` node — more on this later.

2. Why the sphere radius is 1.0 ?

   This is the default radius of spheres produced by `Sphere` node. We could change it by using the `radius` field of a `Sphere` node — more on this later.

Since the material was not specified, the sphere will use the default material properties. These make a light gray diffuse color (expressed as (0.8, 0.8, 0.8) in RGB) and a slight ambient color ((0.2, 0.2, 0.2) RGB).

Figure 1.1. VRML 1.0 sphere example



An equivalent VRML 2.0 file looks like this:

```
#VRML V2.0 utf8

Shape {
  geometry Sphere { }
}
```

As you can see, the header line is now different. It indicates VRML version as 2.0 and encoding as utf8 [1].

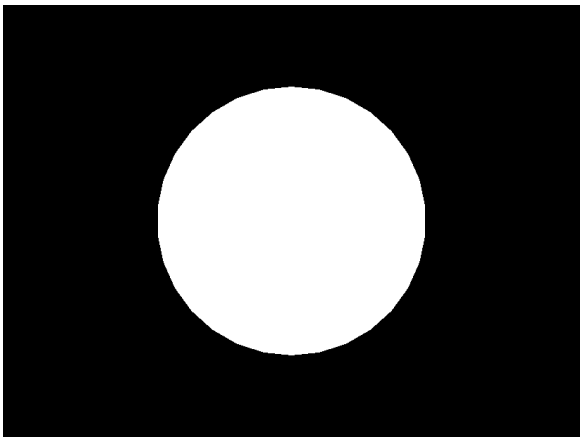
In VRML 2.0 we can't directly use a `Sphere` node. Instead, we have to define a `Shape` node and set it's `geometry` field to our desired `Sphere` node. More on fields and children nodes later.

Actually, our VRML 2.0 example is not equivalent to VRML 1.0 version: in VRML 2.0 version sphere is unlit (it will be rendered using a single white color). It's an example of a general decision in VRML 2.0 specification: *the default behavior is the one that is easiest to render*. If we want to make the sphere lit, we have to add a *material* to it — more on this later.

[1] VRML 2.0 files are always encoded using plain text in utf8. There was a plan to design other encodings, but it was never realized for VRML 2.0. VRML 2.0 files distributed on WWW are often compressed with gzip, we can say that it's a “poor-man's binary encoding”.

X3D (VRML 2.0 successor) filled the gap by specifying three encodings available: “classic VRML encoding” (this is exactly what VRML 2.0 uses), an XML encoding and a binary encoding.

Figure 1.2. VRML 2.0 sphere example



1.2. Fields

Every VRML node has a set of *fields*. A field has a name, a type, and a default value. For example, `Sphere` node has a field named `radius`, of type `SFFloat`, that has a default value of 1.0.

1.2.1. Field types

There are many field types defined by VRML specification. Each field type specifies a syntax for field values in VRML file, and sometimes it specifies some interpretation of the field value. Example field types are:

`SFFloat`	A float value. Syntax is identical to the syntax used in various programming languages, for example `3.1415926` or `12.5e-3`.
`SFLong` (in VRML 1.0), `SFInt32` (in VRML 2.0)	A 32-bit integer value. As you can see, the name was changed in VRML 2.0 to indicate clearly the range of allowed values.
`SFBool`	A boolean value. Syntax: one word, either `FALSE` or `TRUE`. Note that VRML is case-sensitive. In VRML 1.0 you could also write the number 0 (for `FALSE`) or 1 (for `TRUE`), but this additional syntax was removed from VRML 2.0 (since it's quite pointless).
`SFVec2f`, `SFVec3f`	Vector of 2 or 3 floating point values. Syntax is to write them as a sequence of `SFFloat` values, separated by whitespace. The specification doesn't say how these vectors are interpreted: they can be positions, they can be directions etc. The interpretation must be given for each case when some node includes a field of this type.
`SFColor`	Syntax is exactly like `SFVec3f`, but this field has a special interpretation: it's an RGB (red, green, blue) color specification. Each component must be between

0.0 and 1.0. For example, this is a yellow color: `1 1 0`.

`SFRotation` — Four floating point values specifying rotation around an axis. First three values specify an axis, fourth value specifies the angle of rotation (in radians).

`SFImage` — This field type is used to specify image content for `PixelTexture` node in VRML 2.0 (`Texture2` node in VRML 1.0). This way you can specify texture content directly in VRML file, without the need to reference any external file. You can create grayscale, grayscale with alpha, RGB or RGB with alpha images this way. This is sometimes comfortable, when you must include everything in one VRML file, but beware that it makes VRML files very large (because the color values are specified in plain text, and they are not compressed in any way). See VRML specification for exact syntax of this field.

`SFString` — A string, enclosed in double quotes. If you want to include double quote in a string, you have to precede it with the backslash (\) character, and if you want to include the backslash in a string you have to write two backslashes. For example:

```
"This is a string."

"\"To be or not to be\" said the man."

"Windows filename is
   c:\\3dmodels\\tree.wrl"
```

Note that in VRML 2.0 this string can contain characters encoded in utf8 [2].

`SFNode` — This is a special VRML 2.0 field type that contains other node as it's value (or a special value `NULL`). More about this in Section 1.3, “Children nodes”.

All names of field types above start with `SF`, which stands for “single-value field”. Most of these field types have a counterpart, “multiple-value field”, with a name starting with `MF`. For example `MFFloat`, `MFLong`, `MFInt32`, `MFVec2f` and `MFVec3f`. The MF-field value is a sequence of any number (possibly zero) of single field values. For example, `MFVec3f` field specifies any number of 3-component vectors and can be used to specify a set of 3D positions.

Syntax of multiple-value fields is:

1. An opening bracket (`[`).
2. A list of single field values separated by commas (in VRML 1.0) or whitespaces (in VRML 2.0). Note that *in VRML 2.0 comma is also a whitespace*, so if you write commas between values your syntax is valid in all VRML versions.

[2] But also note that my engine doesn't support utf8 yet. In particular, when rendering `Text` node, the string is treated as a sequence of 8-bit characters in ISO-8859-1 encoding.

3. A closing bracket (`]`). Note that you can omit both brackets if your MF-field has exactly one value.

1.2.2. Placing fields within nodes

Each node has a set of fields given by VRML specification. VRML file can specify value of some (maybe all, maybe none) node's fields. You can *always* leave the value of a field unspecified in VRML file, and it *always* is equivalent to explicitly specifying the default value for given field.

VRML syntax for specifying node fields is simple: within node's braces (`{` and `}`) place field's name followed by field's value.

1.2.3. Examples

Let's see some examples of specifying field values.

`Sphere` node has a field named `radius` of type `SFFloat` with a default value 1.0. So the file below is exactly equivalent to our first sphere example in previous section:

```
#VRML V1.0 ascii

Sphere {
  radius 1
}
```

And this is a sphere with radius 2.0:

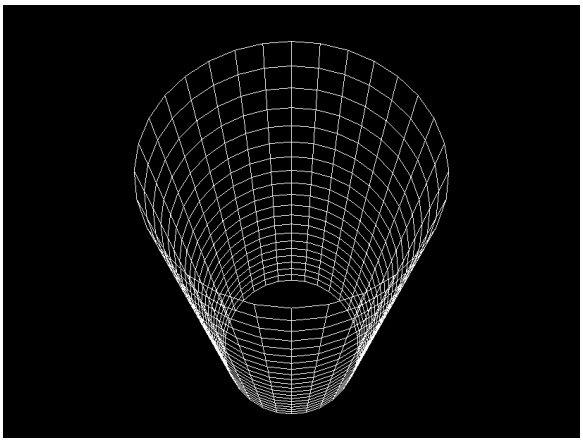
```
#VRML V1.0 ascii

Sphere {
  radius 2
}
```

Here's a VRML 2.0 file that specifies a cylinder that should be rendered without bottom and top parts (thus creating a tube), with a radius 2.0 and height 4.0. Three `SFBool` fields of `Cylinder` are used: `bottom`, `side`, `top` (by default all are `TRUE`, so actually we didn't have to write `side TRUE`). And two `SFFloat` fields, `radius` and `height`, are used.

Remember that in VRML 2.0 we can't just write the `Cylinder` node. Instead we have to use the `Shape` node. The `Shape` node has a field `geometry` of type `SFNode`. By default, value of this field is `NULL`, which means that no shape is defined. We can place our `Cylinder` node as a value of this field to correctly define a cylinder.

```
#VRML V2.0 utf8

Shape {
  geometry Cylinder {
    side TRUE
    bottom FALSE
    top FALSE
    radius 2.0
    height 10.0
  }
}
```

Figure 1.3. Cylinder example, rendered in wireframe mode (because it's unlit, non-wireframe rendering would look confusing)

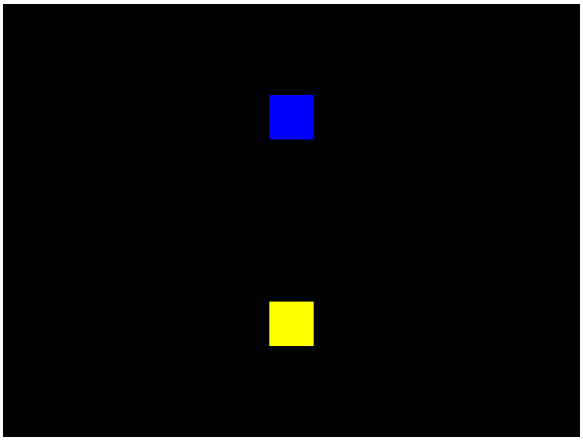


Here's a VRML 2.0 file that specifies two points. Just like in the previous example, we had to use a `Shape` node and place `PointSet` node in it's `geometry` field. `PointSet` node, in turn, has two more `SFNode` fields: `coord` (that can contain `Coordinate` node) and `color` (that can contain `Color` node). `Coordinate` node has a `point` field of type `MFVec3f` — these are positions of defined points. `Color` node has a `color` field of type `MFColor` — these are colors of points, specified in the same order as in the `Coordinate` node.

Note that `PointSet` and `Color` nodes have the same field name: `color`. In the first case, this is an `SFNode` field, in the second case it's an `MFVec3f` field.

```
#VRML V2.0 utf8

Shape {
  geometry PointSet {
    coord Coordinate { point [ 0 -2 0, 0 2 0 ] }
    color Color { color [ 1 1 0, 0 0 1 ] }
  }
}
```

Figure 1.4. VRML points example: yellow point at the bottom, blue point at the top



1.3. Children nodes

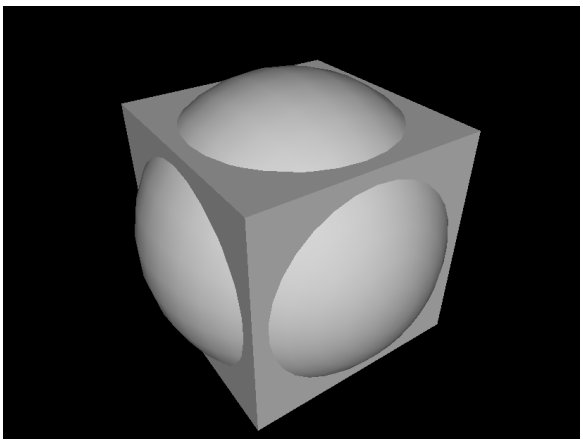
Now we're approaching the fundamental idea of VRML: some nodes can be placed as a children of other nodes. We already saw some examples of this idea in VRML 2.0 examples above: we placed various nodes inside `geometry` field of `Shape` node. VRML 1.0 has a little different way of specifying children nodes (inherited from Inventor format) than VRML 2.0 and X3D — we will see both methods.

1.3.1. Group node examples

In VRML 1.0, you just place children nodes inside the parent node. Like this:

```
#VRML V1.0 ascii

Group {
  Sphere { }
  Cube { width 1.5 height 1.5 depth 1.5 }
}
```

Figure 1.5. A cube and a sphere in VRML 1.0



`Group` is the simplest grouping node. It has no fields, and it's only purpose is just to treat a couple of nodes as one node.

Note that in VRML 1.0 it's required that a whole VRML file consists of exactly one root node, so we actually had to use some grouping node here. For example the following file is invalid according to VRML 1.0 specification:

```
#VRML V1.0 ascii

Sphere { }
Cube { width 1.5 height 1.5 depth 1.5 }
```

Nevertheless the above example is handled by many VRML engines, including our engine described in this document.

In VRML 2.0, you don't place children nodes directly inside the parent node. Instead you place children nodes inside fields of type `SFNode` (this contains zero (`NULL`) or one node) or `MFNode` (this contains any number (possibly zero) of nodes). For example, in VRML 2.0 `Group` node has an `MFNode` field `children`, so the example file in VRML 2.0 equi-

valent to previous example looks like this:

```
#VRML V2.0 utf8

Group {
  children [
    Shape { geometry Sphere { } }
    Shape { geometry Box { size 1.5 1.5 1.5 } }
  ]
}
```

Syntax of `MFNode` is just like for other multiple-valued fields: a sequence of values, inside brackets (`[` and `]`).

Example above also shows a couple of other differences between VRML 1.0 and 2.0:

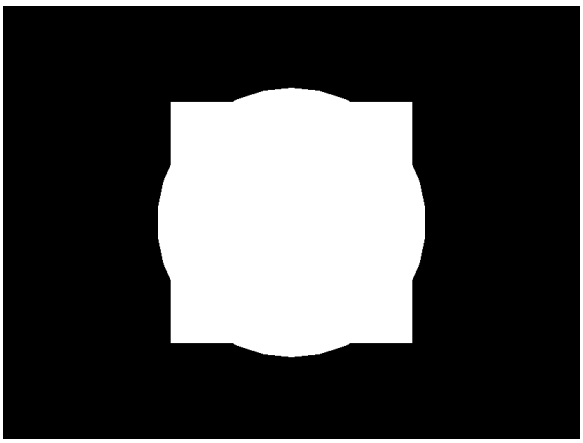
1. In VRML 2.0 we have to wrap `Sphere` and `Box` nodes inside a `Shape` node.
2. Node `Cube` from VRML 1.0 was renamed to `Box` in VRML 2.0.
3. Size of the box in VRML 2.0 is specified using `size` field of type `SFVec3f`, while in VRML 1.0 we had three fields (`width`, `height`, `depth`) of type `SFFloat`.

While we're talking about VRML versions differences, note also that in VRML 2.0 a file can have any number of root nodes. So actually we didn't have to use `Group` node in our example, and the following would be correct VRML 2.0 file too:

```
#VRML V2.0 utf8

Shape { geometry Sphere { } }
Shape { geometry Box { size 1.5 1.5 1.5 } }
```

To be honest, we have to point one more VRML difference: as was mentioned before, in VRML 2.0 shapes are unlit by default. So our VRML 2.0 examples above look like this:

Figure 1.6. An unlit box and a sphere in VRML 2.0



To make them lit, we must assign a *material* for them. In VRML 2.0 you do this by placing a `Material` node inside `material` field of `Appearance` node. Then you place `Appearance` node inside `appearance` field of appropriate `Shape` node. Result looks like this:

```
#VRML V2.0 utf8

Group {
  children [
    Shape {
      appearance Appearance { material Material { } }
      geometry Sphere { }
    }
    Shape {
      appearance Appearance { material Material { } }
      geometry Box { size 1.5 1.5 1.5 }
    }
  ]
}
```

We didn't specify any `Material` node's fields, so the default properties will be used. Default VRML 2.0 material properties are the same as for VRML 1.0: light gray diffuse color and a slight ambient color.

As you can see, VRML 2.0 description gets significantly more verbose than VRML 1.0, but it has many advantages:

1. The way how children nodes are specified in VRML 2.0 requires you to always write an `SFNode` or `MFNode` field name (as opposed to VRML 1.0 where you just write the children nodes). But the advantages are obvious: in VRML 2.0 you can explicitly assign different meaning to different children nodes by placing them within different fields. In VRML 1.0 all the children nodes had to be treated in the same manner — the only thing that differentiated children nodes was their position within the parent.

2. As mentioned earlier, the default behavior of various VRML 2.0 parts is the one that is the easiest to render. That's why the default behavior is to render unlit, and you have to explicitly specify material to get lit objects.

   This is a good thing, since it makes VRML authors more conscious about using features, and hopefully it will force them to create VRML worlds that are easier to render. In the case of rendering unlit objects, this is often perfectly acceptable (or even desired) solution if the object has a detailed texture applied.

3. Placing the `Material` node inside the `SFNode` field of `Appearance`, and then placing the `Appearance` node inside the `SFNode` field of `Shape` may seem like a "bondage-and-discipline language", but it allows various future enhancements of the language without breaking compatibility. For example you could invent a node that allows to specify materials using a different properties (like by describing it's BRDF function, useful for rendering realistic images) and then just allow this node as a value for the `material` field.

   Scenario described above actually happened. First versions of VRML 97 specification didn't include geospatial coordinates support, including a node `GeoCoordinate`. A node `IndexedFaceSet` has a field `coord` used to specify a set of points for geometry, and initially you could place a `Coordinate` node there. When specification of geospatial coordinates support was formulated (and added to VRML 97 specification as optional for VRML browsers), all that had to be changed was to say that now you can place `GeoCoordinate` everywhere where earlier you could use only `Coordinate`.

4. The `Shape` node in VRML 2.0 contains almost whole information needed to render given shape. This means that it's easier to create a VRML rendering engine. We will contrast this with VRML 1.0 approach that requires a lot of state information in Section 1.5, "VRML 1.0 state".

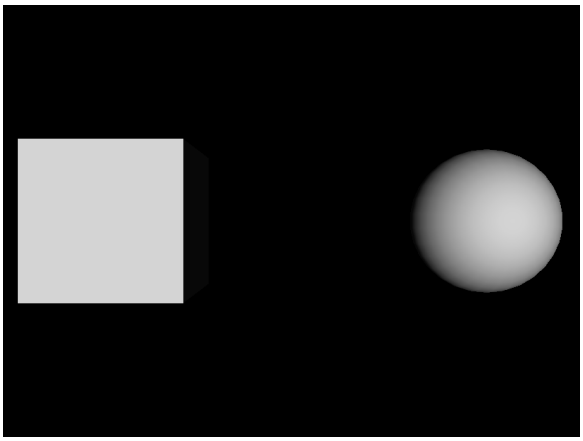
1.3.2. The Transform node

Let's take a look at another grouping node: VRML 2.0 `Transform` node. This node specifies a transformation (a mix of a translation, a rotation and a scale) for all it's children nodes. The default field values are such that no transformation actually takes place, because by default we translate by (0, 0, 0) vector, rotate by zero angle and scale by 1.0 factor. This means that the `Transform` node with all fields left as default is actually equivalent to a `Group` node.

Example of a simple translation:

```
#VRML V2.0 utf8

Shape {
  appearance Appearance { material Material { } }
  geometry Box { }
}

Transform {
  translation 5 0 0
  children Shape {
    appearance Appearance { material Material { } }
    geometry Sphere { }
  }
}
```

Figure 1.7. A box and a translated sphere



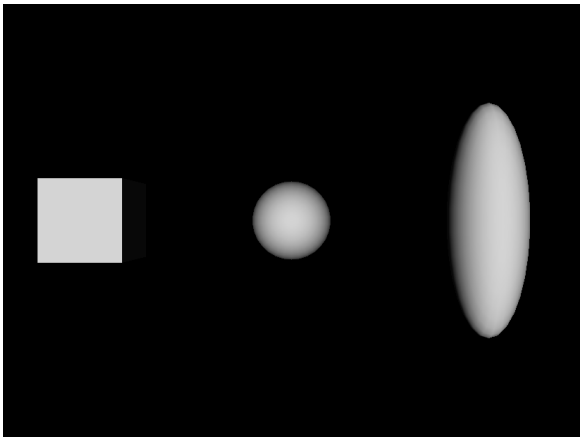
Note that a child of a `Transform` node may be another `Transform` node. All transformations are accumulated. For example these two files are equivalent:

```
#VRML V2.0 utf8

Shape {
  appearance Appearance { material Material { } }
  geometry Box { }
}

Transform {
  translation 5 0 0
  children [
    Shape {
      appearance Appearance { material Material { } }
      geometry Sphere { }
    }
```

```
    Transform {
      translation 5 0 0
      scale 1 3 1
      children Shape {
        appearance Appearance { material Material { } }
        geometry Sphere { }
      }
    }
  ]
}
```

```
#VRML V2.0 utf8

Shape {
  appearance Appearance { material Material { } }
  geometry Box { }
}

Transform {
  translation 5 0 0
  children Shape {
    appearance Appearance { material Material { } }
    geometry Sphere { }
  }
}

Transform {
  translation 10 0 0
  scale 1 3 1
  children Shape {
    appearance Appearance { material Material { } }
    geometry Sphere { }
  }
}
```

Figure 1.8. A box, a translated sphere, and a translated and scaled sphere



1.3.3. Other grouping nodes

- A `Switch` node allows you to choose only one (or none) from children nodes to be in the active (i.e. visible, participating in collision detection etc.) part of the scene. This is

useful for various scripts and it's also useful for hiding nodes referenced later — we will see an example of this in Section 1.4, “DEF / USE mechanism”.

- A `Separator` and a `TransformSeparator` nodes in VRML 1.0. We will see what they do in Section 1.5, “VRML 1.0 state”.

- A `LOD` node (the name is an acronym for *level of detail*) specifies a different versions of the same object. The intention is that all children nodes represent the same object, but with different level of detail: first node is the most detailed one (and difficult to render, check for collisions etc.), second one is less detailed, and so on, until the last node has the least details (it can even be empty, which can be expressed by a `Group` node with no children). VRML browser should choose the appropriate children to render based on the distance between the viewer and designated *center* point.

  Unfortunately, note that LOD is not implemented in our engine yet — for now we always use the most detailed version. As far as VRML specification is concerned, this is acceptable (although non-optimal) behavior.

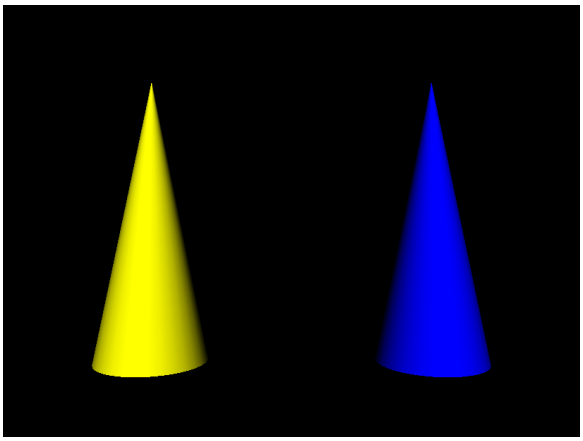
1.4. DEF / USE mechanism

VRML nodes may be named and later referenced. This allows you to reuse the same node (which can be any VRML node type — like a shape, a material, or even a whole group) more than once. The syntax is simple: you name a node by writing `DEF <node-name>` before node type. To reuse the node, just write `USE <node-name>`. This mechanism is available in all VRML versions.

Here's a simple example that uses the same `Cone` twice, each time with a different material color.

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material { diffuseColor 1 1 0 }
  }
  geometry DEF NamedCone Cone { height 5 }
}

Transform {
  translation 5 0 0
  children Shape {
    appearance Appearance { material Material { diffuseColor 0 0 1 } }
    geometry USE NamedCone
  }
}
```

Figure 1.9. Two cones with different materials



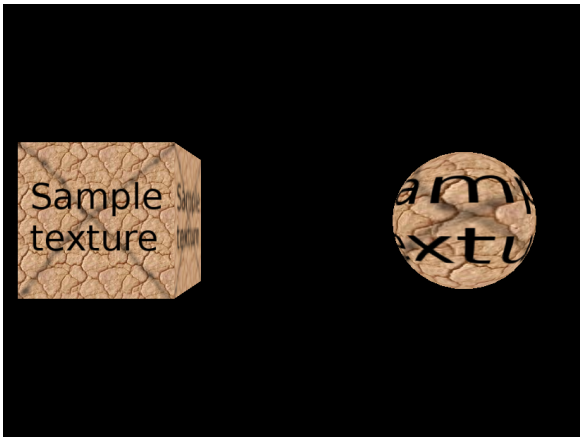
Using DEF / USE mechanism makes your VRML files smaller and easier to author, and it also allows VRML implementations to save resources (memory, loading time...). That's because VRML implementation can allocate the node once, and then just copy the pointer to this node. VRML specifications are formulated to make this approach always correct, even when mixed with features like scripting or sensors. Note that some nodes can “pull” additional data with them (for example `ImageTexture` nodes will load texture image from file), so the memory saving may be even larger. Consider these two VRML files:

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    texture DEF SampleTexture ImageTexture { url "sample_texture.png" }
  }
  geometry Box { }
}

Transform {
  translation 5 0 0
  children Shape {
    appearance Appearance {
      texture USE SampleTexture
    }
    geometry Sphere { }
  }
}
```

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    texture ImageTexture { url "sample_texture.png" }
  }
  geometry Box { }
}

Transform {
  translation 5 0 0
  children Shape {
    appearance Appearance {
      texture ImageTexture { url "sample_texture.png" }
    }
    geometry Sphere { }
  }
```

```
}
```

Figure 1.10. A box and a translated sphere using the same texture



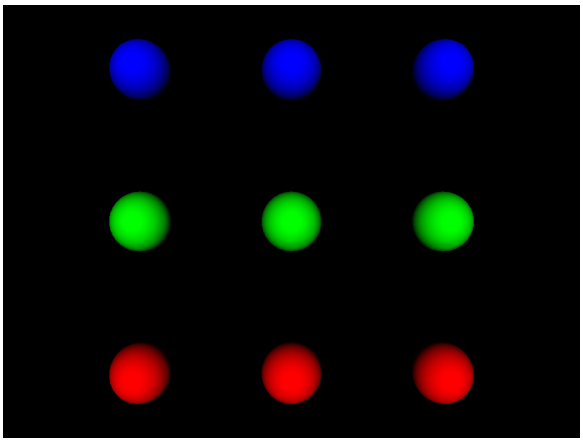
Both files above look the same when rendered, but in the first case VRML implementation loads the texture only once, since we know that this is the same texture node [3].

Note that the first node definition, with `DEF` keyword, not only names the node, but also includes it in the file. Often it's more comfortable to first define a couple of named nodes (without actually using them) and then use them. You can use the `Switch` node for this — by default `Switch` node doesn't include any of it's children nodes, so you can write VRML file like this:

```
#VRML V2.0 utf8

Switch {
  choice [
    DEF RedSphere Shape {
      appearance Appearance { material Material { diffuseColor 1 0 0 } }
      geometry Sphere { }
    }
    DEF GreenSphere Shape {
      appearance Appearance { material Material { diffuseColor 0 1 0 } }
      geometry Sphere { }
    }
    DEF BlueSphere Shape {
      appearance Appearance { material Material { diffuseColor 0 0 1 } }
      geometry Sphere { }
    }
    DEF SphereColumn Group {
      children [
        Transform { translation 0 -5 0 children USE RedSphere }
        Transform { translation 0  0 0 children USE GreenSphere }
        Transform { translation 0  5 0 children USE BlueSphere }
      ]
    }
  ]
}

Transform { translation -5 0 0 children USE SphereColumn }
Transform { translation  0 0 0 children USE SphereColumn }
Transform { translation  5 0 0 children USE SphereColumn }
```

[3] Actually, in the second case, our engine can also figure out that this is the same texture filename and not load the texture twice. But the first case is much “cleaner” and should be generally better for all decent VRML implementations.

Figure 1.11. Three columns of three spheres



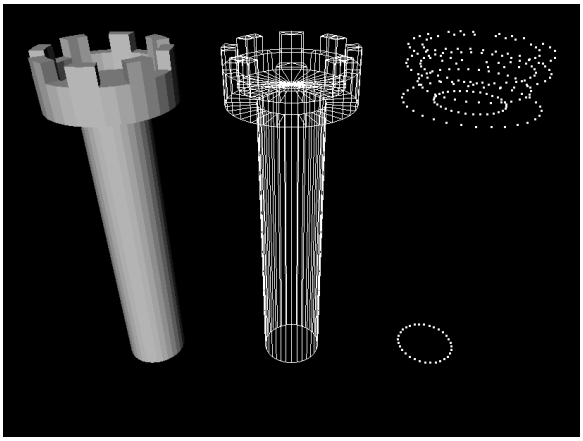
One last example shows a reuse of `Coordinate` node. Remember that a couple of sections earlier we defined a simple `PointSet`. `PointSet` node has an `SFNode` field named `coord`. You can place there a `Coordinate` node. A `Coordinate` node, in turn, has a `point` field of type `SFVec3f` that allows you to specify point positions. The obvious question is "Why all this complexity ? Why not just say that `coord` field is of `SFVec3f` type and directly include the point positions ?". One answer was given earlier when talking about grouping nodes: this allowed VRML specification for painless addition of `GeoCoordinate` as an alternative way to specify positions. Another answer is given by the example below. As you can see, the same set of positions may be used by a couple of different nodes[4].

```
#VRML V2.0 utf8

Switch {
  choice DEF TowerCoordinates Coordinate {
    point [
      4.157832 4.157833 -1.000000,
      4.889094 3.266788 -1.000000,
      ......
    ]
  }
}

Shape {
  appearance Appearance { material Material { } }
  geometry IndexedFaceSet {
    coordIndex [
      63 0 31 32 -1,
      31 30 33 32 -1,
      ......
    ]
    coord USE TowerCoordinates
  }
}

Transform {
  translation 30 0 0
  children Shape {
```

[4] I do not cite full VRML source code here, as it includes a long list of coordinates and indexes generated by Blender exporter. See VRML files distributed with this document: full source is in the file `examples/reuse_coordinate.wrl`.

```
    geometry IndexedLineSet {
      coordIndex [
        63 0 31 32 63 -1,
        31 30 33 32 31 -1,
        ......
      ]
      coord USE TowerCoordinates
    }
  }
}

Transform {
  translation 60 0 0
  children Shape {
    geometry PointSet {
      coord USE TowerCoordinates
    }
  }
}
```

Figure 1.12. Faces, lines and point sets rendered using the same `Coordinate` node



1.4.1. VRML file as a graph

Now that we know all about children relationships and DEF / USE mechanism, we can grasp the statement mentioned at the beginning of this chapter: every VRML file is a directed graph of nodes. It doesn't have cycles, although if we will forget about direction of edges (treat it as an undirected graph), we can get cycles (because of DEF / USE mechanism).

Note that VRML 1.0 file must contain exactly one root node, while VRML 2.0 file is a sequence of any number of root nodes. So, being precise, VRML graph doesn't have to be a connected graph. But actually our engine when reading VRML file with many root nodes just wraps them in an “invisible” `Group` node. This special `Group` node acts just like any other group node, but it's not written back to the file (when e.g. using our engine to pretty-print VRML files). This way, internally, we always see VRML file as a connected graph, with exactly one root node.

1.5. VRML 1.0 state

In previous sections most of the examples were given only in VRML 2.0 version. Partially that's because VRML 2.0 is just newer and better, so you should use it instead of VRML 1.0 whenever possible. But partially that was because we avoided to explain one important behavior of VRML 1.0. In this section we'll fill the gap. Even if you're not interested in VRML 1.0 anymore, this information may help you understand why VRML 2.0 was designed the way it was, and why it's actually better than VRML 1.0. That's because part of the reasons of VRML 2.0 changes were to avoid the whole issue described here.

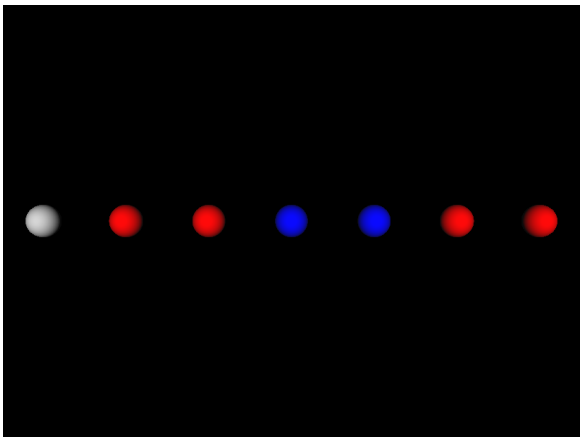
Historically, VRML 1.0 was based on Inventor file format, and Inventor file format was designed specifically with OpenGL implementation in mind. Those of you who do any programming in OpenGL know that OpenGL works as a *state machine*. This means that OpenGL remembers a lot of "global" settings [5]. When you want to render a vertex (aka point) in OpenGL, you just call one simple command (`glVertex`), passing only point coordinates. And the vertex is rendered (along with a line or even a triangle that it produces with other vertexes). What color does the vertex has ? The last color specified by `glColor` call (or `glMaterial`, mixed with lights). What texture coordinate does it have ? Last texture coordinate specified in `glTexCoord` call. What texture does it use ? Last texture bound with `glBindTexture`. We can see a pattern here: when you want to know what property our vertex has, you just have to check what value we last assigned to this property. When we talk about OpenGL state, we talk about all the "last `glColor`", "last `glTexCoord`" etc. values that OpenGL has to remember.

Inventor, and then VRML 1.0, followed a similar approach. "What material does a sphere use ?" The one specified in the last `Material` node. Take a look at the example:

```
#VRML V1.0 ascii

Group {
  # Default material will be used here:
  Sphere { }

  DEF RedMaterial Material { diffuseColor 1 0 0 }

  Transform { translation 5 0 0 }
  # This uses the last material : red
  Sphere { }

  Transform { translation 5 0 0 }
  # This still uses uses the red material
  Sphere { }

  Material { diffuseColor 0 0 1 }

  Transform { translation 5 0 0 }
  # Material changed to blue
  Sphere { }

  Transform { translation 5 0 0 }
  # Still blue...
  Sphere { }

  USE RedMaterial

  Transform { translation 5 0 0 }
  # Red again !
  Sphere { }

  Transform { translation 5 0 0 }
  # Still red.
  Sphere { }
}
```

[5] Actually, they are remembered for each OpenGL context. And, ideally, they are partially "remembered" on graphic board. But we limit our thinking here only to the point of view of a typical program using OpenGL.

Figure 1.13. Spheres with various material in VRML 1.0



Similar answers are given for other questions in the form “What is used ?”. Let's compare VRML 1.0 and 2.0 answers for such questions:

- What texture is used ?

  VRML 1.0 answer: Last `Texture2` node.

  VRML 2.0 answer: Node specified in enclosing `Shape` appearance's `texture` field.

- What coordinates are used by `IndexedFaceSet` ?

  VRML 1.0 answer: Last `Coordinate3` node.

  VRML 2.0 answer: Node specified in `coord` field of given `IndexedFaceSet`.

- What font is used by by `AsciiText` node (renamed to just `Text` in VRML 2.0) ?

  VRML 1.0 answer: Last `FontStyle` node.

  VRML 2.0 answer: Node specified in `fontStyle` field of given `Text` node.

So VRML 1.0 approach maps easily to OpenGL. Simple VRML implementation can just traverse the scene graph, and for each node do appropriate set of OpenGL calls. For example, `Material` node will correspond to a couple of `glMaterial` and `glColor` calls. `Texture2` will correspond to binding prepared OpenGL texture. Visible shapes will cause rendering of appropriate geometry, and so last `Material` and `Texture2` settings will be used.

In our example with materials above you can also see another difference between VRML 1.0 and 2.0, also influenced by the way things are done in OpenGL: the way `Transform` node is used. In VRML 2.0, `Transform` affected it's children. In VRML 1.0, `Transform` node is not supposed to have any children. Instead, it affects *all subsequent nodes*. If we would like to translate last example to VRML 2.0, each `Transform` node would have to be placed as a last child of previous `Transform` node, thus creating a deep nodes hierarchy. Alternatively, we could keep the hierarchy shallow and just use `Transform { translation 5 0 0 ... }` for the first time, then `Transform { translation 10 0 0 ... }`, then `Transform { translation 15 0 0 ... }` and

so on.

This means that simple VRML 1.0 implementation will just call appropriate matrix transformations when processing `Transform` node. In VRML 1.0 there are even more specialized transformation nodes. For example a node `Translation` that has a subset of features of full `Transform` node: it can only translate. Such `Translation` has an excellent, trivial mapping to OpenGL: just call `glTranslate`.

There's one more important feature of OpenGL “state machine” approach: stacks. OpenGL has a matrix stack (actually, three matrix stacks for each matrix type) and an attributes stack. As you can guess, there are nodes in VRML 1.0 that, when implemented in an easy way, map perfectly to OpenGL push/pop stack operations: `Separator` and `TransformSeparator`. When you use `Group` node in VRML 1.0, the properties (like last used `Material` and `Texture2`, and also current transformation and texture transformation) “leak” outside of `Group` node, to all subsequent nodes. But when you use `Separator`, they do not leak out: all transformations and “who's the last material/texture node” properties are unchanged after we leave `Separator` node. So simple `Separator` implementation in OpenGL is trivial:

1. At the beginning, use `glPushAttrib` (saving all OpenGL attributes that can be changed by VRML nodes) and `glPushMatrix` (for both modelview and texture matrices).

2. Then process all children nodes of `Separator`.

3. Then restore state by `glPopAttrib` and `glPopMatrix` calls.

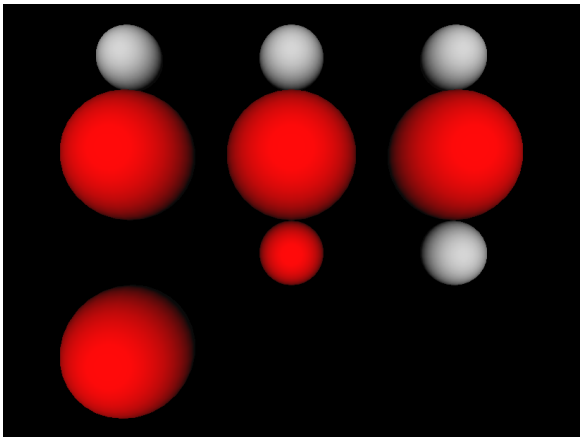
`TransformSeparator` is a cross between a `Separator` and a `Group`: it saves only transformation matrix, and the rest of the state can “leak out”. So to implement this in OpenGL, you just call `glPushMatrix` (on modelview matrix) before processing children and `glPopMatrix` after.

Below is an example how various VRML 1.0 grouping nodes allow “leaking”. Each column starts with a standard `Sphere` node. Then we enter some grouping node (from the left: `Group`, `TransformSeparator` and `Separator`). Inside the grouping node we change material, apply scaling transformation and put another `Sphere` node — middle row always contains a red large sphere. Then we exit from grouping node and put the third `Sphere` node. How does this sphere look like depends on used grouping node.

```
#VRML V1.0 ascii

Separator {
  Sphere { }
  Transform { translation 0 -3 0 }
  Group {
    Material { diffuseColor 1 0 0 }
    Transform { scaleFactor 2 2 2 }
    Sphere { }
  }
  # This was a Group, so both Material change and scaling "leaks out"
  Transform { translation 0 -3 0 }
  Sphere { }
}

Transform { translation 5 0 0 }

Separator {
  Sphere { }
  Transform { translation 0 -3 0 }
  TransformSeparator {
```

```
    Material { diffuseColor 1 0 0 }
    Transform { scaleFactor 2 2 2 }
    Sphere { }
  }
  # This was a TransformSeparator, so only Material change "leaks out"
  Transform { translation 0 -3 0 }
  Sphere { }
}

Transform { translation 5 0 0 }

Separator {
  Sphere { }
  Transform { translation 0 -3 0 }
  Separator {
    Material { diffuseColor 1 0 0 }
    Transform { scaleFactor 2 2 2 }
    Sphere { }
  }
  # This was a Separator, so nothing "leaks out".
  # The last sphere is identical to the first one.
  Transform { translation 0 -3 0 }
  Sphere { }
}
```

Figure 1.14. An example how properties “leak out” from various grouping nodes in VRML 1.0



1.5.1. Why VRML 2.0 is better

There are some advantages of VRML 1.0 “state” approach:

1. It maps easily to OpenGL.

   Such easy mapping may be also quite efficient. For example, if two nodes use the same `Material` node, we can just change OpenGL material once (at the time `Material` node is processed). VRML 2.0 implementation must remember last set `Material` node to achieve this purpose.

2. It's flexible. The way transformations are specified in VRML 2.0 forces us often to create deeper node hierarchies than in VRML 1.0.

And in VRML 1.0 we can easier share materials, textures, font styles and other properties among a couple of nodes. In VRML 2.0 such reusing requires naming nodes by DEF / USE mechanism. In VRML 1.0 we can simply let a couple of nodes have the same node as their last `Material` (or similar) node.

But there are also serious problems with VRML 1.0 approach, that VRML 2.0 solves.

1. The argumentation about “flexibility” of VRML 1.0 above looks similar to argumentation about various programming languages [6], that are indeed flexible but also allow the programmer to “shoot himself in the foot”. It's easy to forget that you changed some material or texture, and accidentally affect more than you wanted.

   Compare this with the luxury of VRML 2.0 author: whenever you start writing a `Shape` node, you always start with a clean state: if you don't specify a texture, shape will not be textured, if you don't specify a material, shape will be unlit, and so on. If you want to know how given `IndexedFaceSet` will look like when rendered, you just have to know it's enclosing `Shape` node. More precisely, the only things that you have to know for VRML 2.0 node to render it are

   - enclosing `Shape` node,
   - accumulated transformation from `Transform` nodes,
   - and some “global” properties: lights that affect this shape and fog properties. I call them “global” because usually they are applied to the whole scene or at least large part of it.

   On the other hand, VRML 1.0 author or reader (human or program) must carefully analyze the code before given node, looking for last `Material` node occurrence etc.

2. The argumentation about “simple VRML 1.0 implementation” misses the point that such simple implementation will in fact suffer from a couple of problems. And fixing these problems will in fact force this implementation to switch to non-trivial methods. The problems include:

   - OpenGL stacks sizes are limited, so a simple implementation will limit allowed depth of `Separator` and `TransformSeparator` nodes.
   - If we will change OpenGL state each time we process a state-changing node, then we can waste a lot of time and resources if actually there are no shapes using given property. For example this code

     ```
     Separator {
       Texture2 { filename "texture.png" }
     }
     ```

     will trick a naive implementation into loading from file and then loading to OpenGL context a completely useless texture data.

     This seems like an irrelevant problem, but it will become a large problem as soon as we will try to use any technique that will have to render only parts of the scene. For example, implementing material transparency using OpenGL blending requires that first all non-transparent shapes are rendered. Also implementing culling of objects to a camera frustum will make many shape nodes in the scene ignored in some frames.

[6] ...programming languages that should remain nameless here...

3. Last but not least: in VRML 1.0, grouping nodes *must* process their children in order, to collect appropriate state information needed to render each shape. In VRML 2.0, there is no such requirement. For example, to render a `Group` node in VRML 2.0, implementation can process and render children nodes in any order. Like said above, VRML 2.0 must only know about current transformation and global things like fog and lights. The rest of information needed is always contained within appropriate `Shape` node.

   VRML 2.0 implementation can even ignore some children in `Group` node if it's known that they are not visible.

   Example situations when implementation should be able to freely choose which shapes (and in what order) are rendered were given above: implementing transparency using blending, and culling to camera frustum.

   More about the way how we solved this problem for both VRML 1.0 and 2.0 in Section 2.10, “Flat scene”. More about OpenGL blending and culling to frustum in Section 5.3, “Flat scene for OpenGL”.

1.6. Other important VRML features

Now that we're accustomed with VRML syntax and concepts, let's take a quick look at some notable VRML features that weren't shown yet.

1.6.1. Inline nodes

A powerful tool of VRML is the ability to include one model as a part of another. In VRML 2.0 we do this by `Inline` node. It's `url` field specifies the URL (possibly relative) of VRML file to load. Note that our engine doesn't actually support URLs right now and treats this just as a file name.

The content of referenced VRML file is placed at the position of given `Inline` node. This means that you can apply transformation to inlined content. This also means that including the same file more than once is sensible in some situations. But remember the remarks in Section 1.4, “DEF / USE mechanism”: if you want to include the same file more than once, you should name the `Inline` node and then just reuse it. Such reuse will conserve resources.

`url` field is actually `MFString` and is a sequence of URL values, from the most to least preferred one. So VRML browser will try to load files from given URLs in order, until a valid file will be found.

In VRML 1.0 the node is called `WWWInline`, and the URL (only one is allowed, it's `SFString` field) is specified in the field `name`.

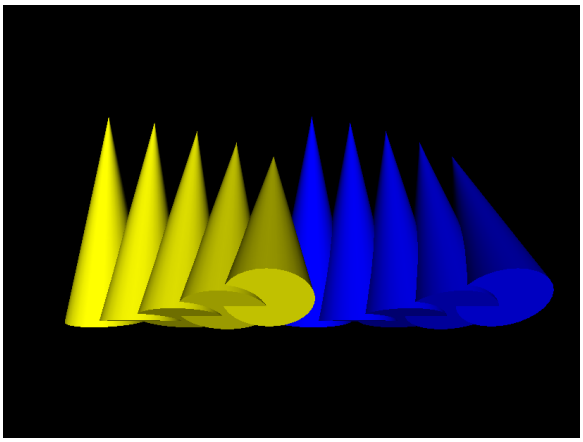
When using our engine you can mix VRML versions and include VRML 1.0 file from VRML 2.0, or the other way around. Moreover, you can include 3DS and Wavefront OBJ files too.

An example:

```
#VRML V2.0 utf8

DEF MyInline Inline { url "reuse_cone.wrl" }

Transform {
  translation 1 0 0
```

```
  rotation 1 0 0 -0.2
  children [
    USE MyInline

Transform {
  translation 1 0 0
  rotation 1 0 0 -0.2
  children [
    USE MyInline

Transform {
  translation 1 0 0
  rotation 1 0 0 -0.2
  children [
    USE MyInline

Transform {
  translation 1 0 0
  rotation 1 0 0 -0.2
  children [
    USE MyInline

] } ] } ] } ] }
```

Figure 1.15. Our earlier example of reusing cone inlined a couple of times, each time with a slight translation and rotation



1.6.2. Texture transformation

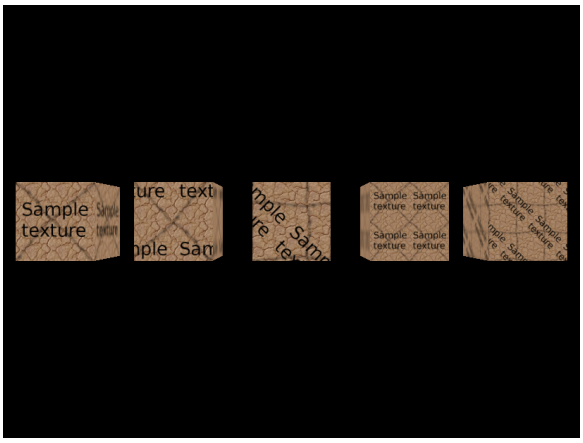
VRML allows you to specify a texture coordinate transformation. This allows you to translate, scale and rotate visible texture on given shape.

In VRML 1.0, you do this by `Texture2Transform` node — this works analogous to `Transform`, but transformations are only 2D. Texture transformations in VRML 1.0 accumulate, just like normal transformations. Here's an example:

```
#VRML V1.0 ascii

Group {
  Texture2 { filename "sample_texture.png" }

  Cube { }

  Transform { translation 3 0 0 }
```

```
  Separator {
    # translate texture
    Texture2Transform { translation 0.5 0.5 }
    Cube { }
  }

  Transform { translation 3 0 0 }

  Separator {
    # rotate texture by Pi/4
    Texture2Transform { rotation 0.7853981634 }
    Cube { }
  }

  Transform { translation 3 0 0 }

  Separator {
    # scale texture
    Texture2Transform { scaleFactor 2 2 }
    Cube { }

    Transform { translation 3 0 0 }

    # rotate texture by Pi/4.
    # Texture transformation accumulates, so this will
    # be both scaled and rotated.
    Texture2Transform { rotation 0.7853981634 }
    Cube { }
  }
}
```

Figure 1.16. Textured cube with various texture transformations



Remember that we transform *texture coordinates*, so e.g. scale 2x means that the texture appears *2 times smaller*.

VRML 2.0 proposes a different approach here: We have similar `TextureTransform` node, but we can use it only as a value for `textureTransform` field of `Appearance`. This also means that there is no way how texture transformations could accumulate. Here's a VRML 2.0 file equivalent to previous VRML 1.0 example:

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    texture DEF SampleTexture ImageTexture { url "sample_texture.png" }
```

```
  }
  geometry Box { }
}

Transform {
  translation 3 0 0
  children Shape {
    appearance Appearance {
      texture USE SampleTexture
      # translate texture
      textureTransform TextureTransform { translation 0.5 0.5 }
    }
    geometry Box { }
  }
}

Transform {
  translation 6 0 0
  children Shape {
    appearance Appearance {
      texture USE SampleTexture
      # rotate texture by Pi/4
      textureTransform TextureTransform { rotation 0.7853981634 }
    }
    geometry Box { }
  }
}

Transform {
  translation 9 0 0
  children Shape {
    appearance Appearance {
      texture USE SampleTexture
      # scale texture
      textureTransform TextureTransform { scale 2 2 }
    }
    geometry Box { }
  }
}

Transform {
  translation 12 0 0
  children Shape {
    appearance Appearance {
      texture USE SampleTexture
      # scale and rotate the texture.
      # There's no way to accumulate texture transformations,
      # so we just do both rotation and scaling by
      # TextureTransform node below.
      textureTransform TextureTransform {
        rotation 0.7853981634
        scale 2 2
      }
    }
    geometry Box { }
  }
}
```

1.6.3. Navigation

You can specify various navigation information using the `NavigationInfo` node.

- `type` field describes preferred navigation type. You can “EXAMINE” model, “WALK” in the model (with collision detection and gravity) and “FLY” (collision detection, but no gravity).

- `avatarSize` field sets viewer (avatar) sizes. These typically have to be adjusted for each world to "feel right". Although you should note that VRML generally suggests to treat length 1.0 in your world as "1 meter". If you will design your VRML world following this assumption, then default `avatarSize` will feel quite adequate, assuming that you want the viewer to have human size in your world. Viewer sizes are used for collision detection.

- Viewer size together with `visibilityLimit` may be also used to set VRML browsers Z-buffer near and far clipping planes. This is the case with our engine. By default our engine tries to calculate sensible values for near and far based on scene bounding box size.

- You can also specify moving speed (`speed` field), and whether head light is on (`headlight` field).

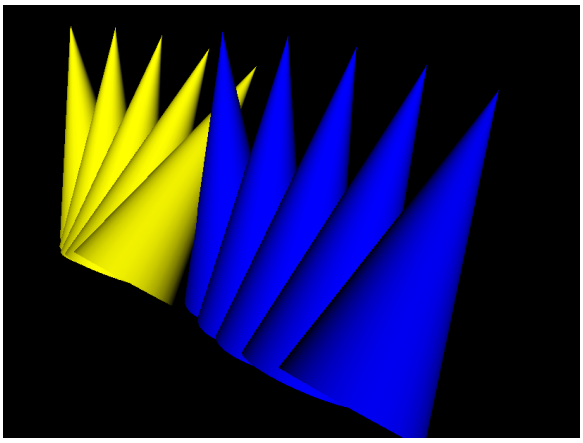
To specify default viewer position and orientation in the world you use `Viewpoint` node. In VRML 1.0, instead of `Viewpoint` you have `PerspectiveCamera` and `OrthogonalCamera` (in VRML 2.0 viewpoint is always perspective). Viewpoint and camera nodes may be generally specified anywhere in the file. The first viewpoint/camera node found in the file (but only in the active part of the file — e.g. not in inactive children of `Switch`) will be used as the starting position/orientation. Note that viewpoint/camera nodes are also affected by transformation.

Finally, note that my VRML viewer view3dscene [http:// www. camelot. homedns. org/ ~michalis/view3dscene.php] has a useful function to print VRML viewpoint/camera nodes ready to be pasted to VRML file, see menu item "Console" -> "Print current camera node".

Here's an example file. It defines a viewpoint (generated by `view3dscene`) and a navigation info and then includes actual world geometry from other file (shown in our earlier example about inlining).

```
#VRML V2.0 utf8

Viewpoint  {
  position 11.832 2.897 6.162
  orientation -0.463 0.868 0.172 0.810
}

NavigationInfo {
  avatarSize [ 0.5, 2 ]
  speed 1.0
  headlight TRUE
}

Inline { url "inline.wrl" }
```

Figure 1.17. Viewpoint defined for our previous example with multiplied cones



1.6.4. IndexedFaceSet features

`IndexedFaceSet` nodes (and a couple of other nodes in VRML 2.0 like `ElevationGrid`) have some notable features to make their rendering better and more efficient:

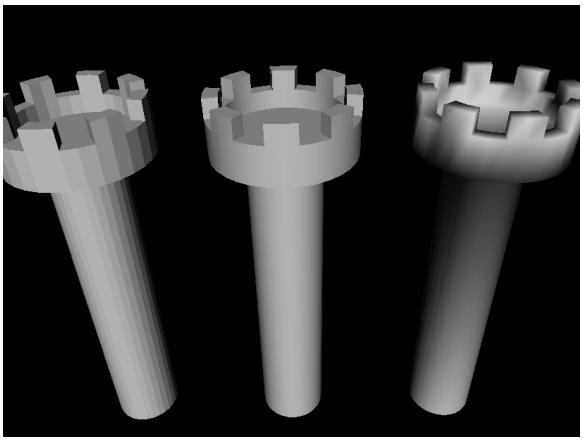
- You can use non-convex faces if you set `convex` field to `FALSE`. It will be VRML browser's responsibility to correctly triangulate them. By default faces are assumed to be convex (following the general rule that the default behavior is the easiest one to handle by VRML browsers).
- By default shapes are assumed to be `solid` which allows to use backface culling when rendering them.
- If you don't supply pre-generated normal vectors for your shapes, they will be calculated by the VRML browser. You can control how they will be calculated by the `creaseAngle` field: if the angle between adjacent faces will be less than specified `creaseAngle`, the normal vectors in appropriate points will be smooth. This allows you to specify preferred “smoothness” of the shape. In VRML 2.0 by default `creaseAngle` is zero (so all normals are flat; again this follows the rule that the default behavior is the easiest one for VRML browsers). See example below.
- For VRML 1.0 the `creaseAngle`, backface culling and convex faces settings are controlled by `ShapeHints` node.
- All VRML shapes have some sensible default texture mapping. This means that you don't have to specify texture coordinates if you want the texture mapped. You only have to specify some texture. For `IndexedFaceSet` the default texture mapping adjusts to shape's bounding box (see VRML specification for details).

Here's an example of the `creaseAngle` use. Three times we define the same geometry in `IndexedFaceSet` node, each time using different `creaseAngle` values. The left tower uses `creaseAngle 0`, so all faces are rendered flat. Second tower uses `creaseAngle 1` and it looks good — smooth where it should be. The third tower uses `creaseAngle 4`, which just means that normals are smoothed everywhere (this case is actually optimized inside our engine, so it's calculated faster) — it looks bad, we can see

that normals are smoothed where they shouldn't be.

```
#VRML V2.0 utf8

Viewpoint  {
  position 31.893 -69.771 89.662
  orientation 0.999 0.022 -0.012 0.974
}

Switch {
  choice DEF TowerCoordinates Coordinate {
    point [
      4.157832 4.157833 -1.000000,
      4.889094 3.266788 -1.000000,
      ........
    ]
  }
}

Transform {
  children Shape {
    appearance Appearance { material Material { } }
    geometry IndexedFaceSet {
      coordIndex [
        63 0 31 32 -1,
        31 30 33 32 -1,
        ........
      ]
      coord USE TowerCoordinates
      creaseAngle 0
    }
  }
}

Transform {
  translation 30 0 0
  children Shape {
    appearance Appearance { material Material { } }
    geometry IndexedFaceSet {
      coordIndex [
        63 0 31 32 -1,
        31 30 33 32 -1,
        ........
      ]
      coord USE TowerCoordinates
      creaseAngle 1
    }
  }
}

Transform {
  translation 60 0 0
  children Shape {
    appearance Appearance { material Material { } }
    geometry IndexedFaceSet {
      coordIndex [
        63 0 31 32 -1,
        31 30 33 32 -1,
        ........
      ]
      coord USE TowerCoordinates
      creaseAngle 4
    }
  }
}
```

Figure 1.18. Three towers with various `creaseAngle` settings



1.6.5. Beyond what is implemented

There are some very notable VRML 97 features that I didn't describe in this document, simply because they are not implemented yet. To name just a few:

Interpolator nodes

They define animation by interpolation of appropriate sets of values. My engine also allows to do animations, also by interpolating, so the internal approach is actually the same. However the way to specify the animations for my engine is not by VRML nodes, but instead by providing two or more VRML models with the same structure. My approach has some advantages and some disadvantages when compared to VRML interpolators, the details will be explained in Chapter 6, *Animation*.

Prototypes

These constructions define new VRML nodes in terms of already available ones. The idea is basically like macros, but it works on VRML nodes level (not on textual level, even not on VRML tokens level) so it's really safe.

External prototypes

These constructions define syntax of new VRML nodes, without defining their implementation. The implementation can be specified in other VRML file (using normal prototypes mentioned above) or can be deduced by particular VRML browser using some browser-specific means (for example, a browser may just have some non-standard nodes built-in). If a browser doesn't know how to handle given node, it can at least correctly parse the node (and ignore it).

For example, many VRML browsers handle some non-standard VRML nodes. If you use these nodes and you want to make your VRML files at least readable by other VRML browsers, you should declare these non-standard nodes using external prototypes.

My engine doesn't handle external prototype constructions

yet[7]. When my engine approaches unknown node type it is able to ignore it without really parsing it's fields: it just parses the file up to the matching brace token. So it doesn't actually need the external prototype.

Sensors, events, scripting

VRML 97 specification includes a great support for extending content with any kind of external language. Detailed description of Java and ECMAScript (JavaScript) bindings is given by the specification, and it's expected that other languages could use similar approaches. X3D specification pushes this even further, by describing external language interface in a way that is “neutral” to actual programming language (which means that it should be applicable to pretty much all existing programming languages).

Scripts can be invoked on various events, and the events can in turn be generated by various nodes. In particular, sensor nodes are special kind of nodes that were designed only to generate events in particular situations.

My engine doesn't support any kind of scripting for now. My initial approach was directed at making special programs (like games ...) that simply use the VRML engine, so any logic was expressed in normal ObjectPascal code that was later compiled.

Of course, it would be great to implement scripting and move as much of this logic as possible to VRML files. For VRML authors, this is also the way to not be tied to any particular VRML engine. Although for really large programs there's no way that whole logic could be moved into a scripting language...

NURBS

NURBS curves and surfaces. Optional in VRML 97 specification.

[7]Although I already handle a couple of non-standard VRML extensions, see my extensions list [http://www.camelot.homedns.org/~michalis/kambi_vrml_extensions.php]

Chapter 2. Reading, writing, processing VRML scene graph

This and the following chapters will describe how our VRML engine works. We will describe used data structures and algorithms. Together this should give you a good idea of what our engine is capable of, where are it's strengths and weaknesses, and how it's all achieved.

In this document we should *not* go into details about some ObjectPascal-specific language constructs or solutions — this would be too low-level stuff, uninteresting from a general point of view. If you're an ObjectPascal programmer and you want to actually use my engine then you may find it helpful to study source code [http:// camelot. homedns. org/ ~michalis/ sources. php] (especially example programs in `examples` subdirectories) and units documentation [http://camelot.homedns.org/~michalis/sources_docs.php] while reading this document. If you only want to read this document, everything that you need is some basic idea about object-oriented programming.

2.1. TVRMLNode class basics

The base class of our engine is the `TVRMLNode` class, not surprisingly representing a VRML node. This is an abstract class, for all specific VRML node types we have some descendant of `TVRMLNode` defined. Naming convention for non-abstract node classes is like `TNodeCoordinate` class for VRML `Coordinate` node type.

Every VRML node has it's fields available in it's `Fields` property. You can also access individual fields by properties named like `FdXxx`, for example `FdPoint` is a property of `TNodeCoordinate` class that represents `point` field of `Coordinate` node.

VRML 1.0 children nodes are accessed by `Children` and `ChildrenCount` properties. For VRML 2.0 this is not needed, since you access all children nodes by accessing appropriate `SFNode` and `MFNode` fields. A convenience properties named `SmartChildren` and `SmartChildrenCount` are defined: for “normal” VRML 2.0 grouping nodes (this mostly means nodes with `MFNode` field named `children`) the `SmartChildrenXxx` properties operate on appropriate `MFNode`, for other nodes they operate on VRML 1.0 `ChildrenXxx` properties.

Because of DEF / USE mechanism each node may be a children (“children” both in the VRML 1.0 and 2.0 senses) of more than one node. This means that we cannot use some trivial destructing strategy. When we destruct some node's instance, we cannot simply destruct all it's children, because they are possibly used in other nodes. The simple solution to this is to keep track in each node about it's parents. Each node has properties `ParentNodes` and `ParentNodesCount` that track information about all the nodes that use it in VRML 1.0 style (i.e. on `TVRMLNode.Children` list). And properties `ParentFields` and `ParentFieldsCount` that track information about all the `SFNode` and `MFNode` fields referencing this node. The children node is automatically destroyed when it has no parents — which means that both `ParentNodesCount` and `ParentFieldsCount` are zero. Effectively, we implemented *reference-counting*. And as a bonus, `ParentXxx` properties are sometimes helpful when we want to do some “bottom-to-top” processing of VRML graph (although this should be generally avoided, “top-to-bottom” processing is much more in the spirit of the VRML graph).

Classes for VRML nodes specific to particular VRML version get a suffix `_1` or `_2` representing their intended VRML version. For example, we have `TNodeIndexedFace-`

`Set_1` (for VRML 1.0) and `TNodeIndexedFaceSet_2` (for VRML 2.0) classes. Such nodes always have their `ForVRMLVersion` method overridden to indicate in what VRML version they are allowed to be used. For example, when parser starts reading `IndexedFaceSet` node, it creates either `TNodeIndexedFaceSet_1` or `TNodeIndexedFaceSet_2`, depending on VRML version indicated in the file header line. Note that this separation between VRML versions is done only when reading VRML nodes from file. When processing VRML nodes graph by code you can freely mix VRML nodes from various VRML versions and everything will work, including writing nodes back to VRML file (although if you mix VRML versions too carelessly you may get VRML file that can only be read back by my engine, and not by other engines that may be limited to only VRML 1.0 or only VRML 2.0). More on this later in Section 2.2, “The sum of VRML 1.0 and 2.0”.

The result of parsing any VRML file is always a single `TVRMLNode` instance representing the root node of the given file. If the file had more than one root node [1] then our engine wraps them in an additional `Group` node. More precisely, additional instance of `TNodeGroupHidden_1` or `TNodeGroupHidden_2` is created. They descend from `TNodeGroup_1` and `TNodeGroup_2`, accordingly, and so can be always treated as 100% normal `Group` nodes. At the same time, VRML writing code can take special precautions to not record these “fake” group nodes back to VRML file.

2.2. The sum of VRML 1.0 and 2.0

Our engine handles both VRML 1.0 and VRML 2.0. As we have seen in Chapter 1, *Overview of VRML*, there are important differences between these VRML versions. The way how I decided to handle both VRML versions is the more difficult, but also more complete approach. Effectively, you have the *sum of VRML 1.0 and 2.0 features available*.

I decided to avoid trying to create some internal conversions from VRML 1.0 to VRML 2.0, or VRML 2.0 to 1.0, or to some newly invented internal format. I wanted to have a full, flexible, 100% conforming to VRML 1.0 and VRML 2.0 specifications engine. And the fact is that any conversion along the way will likely cause problems — ideologically speaking, that's because there is always something lost, or at least difficult to recover, when a complicated conversion is done.

Practically here are some reasons why a simple conversion between VRML 1.0 and VRML 2.0 is not possible, in any direction:

1. VRML 2.0 specification authors intentionally wanted to simplify some things that people (both VRML world authors and VRML browser implementors) thought were unnecessarily complicated in VRML 1.0. This causes problems for a potential converter from VRML 1.0 to 2.0, since it will have trouble to express some VRML 1.0 constructs. For example:

   - In VRML 1.0 you can specify multiple materials for a single geometry node. In VRML 2.0 each geometry node uses at most one material. So a potential converter from VRML 1.0 to 2.0 may need to split geometry nodes.

   - In VRML 1.0 you can accumulate texture transformations (`Texture2Transform` nodes). In VRML 2.0 you can't (you can only place one `TextureTransform` node in the `Appearance.textureTransform` field). So a potential converter must

[1] Multiple root nodes are allowed in VRML 2.0 specification. Our engine also allows them for VRML 1.0 because it's an extension often expected by VRML 1.0 creators (humans and programs).

accumulate texture transformations on it's own. And this is not trivial in a general case, because you can't directly specify texture transformation matrix in VRML 2.0. Instead you have to express texture transformation in terms of one translation, one rotation and one scaling.

- In VRML 1.0 you can specify any 4x4 matrix transformation using `MatrixTransformation` node. This is not possible at all in VRML 2.0. In VRML 2.0 geometry transformation must be specified in terms of translations, rotations and scaling.

- In VRML 1.0 you can limit which geometry nodes are affected by `PointLight` or `SpotLight` by placing light nodes at particular points in the node hierarchy. That's because in VRML 1.0 light nodes work just like other “state changing” nodes: they affect all subsequent nodes, until blocked by the end of the `Separator` node.

  In VRML 2.0 this doesn't work. You cannot control what parts of the scene are affected by light nodes by placing light nodes at some particular place in the node hierarchy. Instead, you have to use `radius` field of light nodes. This means that some VRML 1.0 tricks are simply not possible.

- `OrthographicCamera` is not possible to express using VRML 2.0 standard nodes.

Summary: in certain cases translating VRML 1.0 to 2.0 can be very hard or even impossible. If we want to handle VRML 1.0 perfectly, we can't just write a converter from VRML 1.0 to 2.0 and then define every operation only in terms of VRML 2.0.

2. On the other hand, VRML 2.0 also includes various things not present in VRML 1.0. This includes many new nodes, that often cannot be expressed at all in VRML 1.0: all sensors, scripts, interpolators, special things like `Collision` and `Billboard`.

   Moreover, VRML 2.0 uses `SFNode` (with possible `NULL` value) and `MFNode`, and generally reduces the state that needs to be remembered when processing VRML graph. This means that many existing features have to be expressed differently.

   For example consider specifying normals for `IndexedFaceSet`. In VRML 2.0 everything that decides about how generated normals are supplied are the `normal` and `normalIndex` fields of given `IndexedFaceSet` node. We take advantage of the `SFNode` field type, and say that whole `Normal` node may be just placed within `normal` field of `IndexedFaceSet`. So we just keep whole knowledge inside `IndexedFaceSet` node.

   On the other hand, in VRML 1.0 we have to use the value of last `NormalBinding` node. This says whether we should use the last `Normal` node, and how.

   Potential VRML 2.0 to 1.0 converter would have to make a lot of effort to “deconstruct” VRML 2.0 shape properties back to VRML 1.0 state nodes. This makes conversion difficult to revert (e.g. when we want to write VRML 2.0 content back to file).

That's why I decided to support in my engine the sum of all VRML features. For example, VRML 1.0 nodes can have direct children nodes, so I support it (by `Children` property of `TVRMLNode`). VRML 2.0 nodes can have children nodes through `SFNode` and `MFNode` fields, so I support it too. I'm not trying hard to "combine" these two ideas (direct children nodes and children inside `MFNode`) into one — I just implement and handle them both [2].

In some cases this approach forces me to do more work. For example, for many routines that calculate bounding boxes of shape nodes, I had to prepare three routines:

1. Common implementation, as a static procedure inside the `VRMLNodes` unit. This handles actual calculation and as parameters expects already calculated properties of given shape node. As a simple example, when calculating bounding box of a cube, we expect to get three parameters describing cube's sizes in X, Y and Z dimension.

2. VRML 1.0 implementation in VRML 1.0-specific node version that calls the common implementation, after preparing parameters for common implementation. As a simple example, `TNodeCube_1` (VRML 1.0 cube shape) just uses it's `FdWidth`, `FdHeight` and `FdDepth` as appropriate sizes.

3. And VRML 2.0 implementation in VRML 2.0-specific node version, that also calls the common implementation after preparing it's parameters. As a simple example, `TNodeBox` (VRML 2.0 cube shape) accesses three items of it's `FdSize` field to get the appropriate sizes.

In our simple example above we talked about a cube shape, and the whole issue with calculating three size values differently for VRML 1.0 and 2.0 was actually trivial. But the point is that for some nodes, like `IndexedFaceSet`, this is much harder.

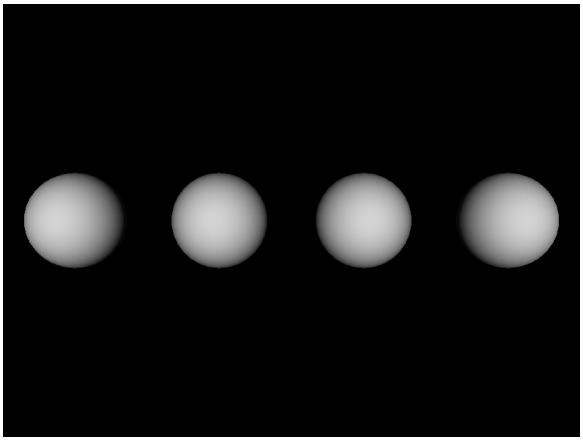
For VRML authors this "sum" approach means that when reading VRML 1.0, many VRML 2.0 constructs (that not conflict with anything in VRML 1.0) are allowed, and the other way around too. That's why you can actually mix VRML 1.0 and 2.0 code in my engine. Consider this strange file:

```
#VRML V2.0 utf8

Separator {
  DEF VRML2Cube Shape {
    appearance Appearance { material Material { } }
    geometry Sphere { }
  }

  Translation { translation 3 0 0 }

  USE VRML2Cube

  Transform {
    translation 3 0 0
    children [
      USE VRML2Cube

      Translation { translation 3 0 0 }

      USE VRML2Cube
    ]
  }
```

[2] `SmartChildrenXxx` properties mentioned in the previous section somewhat combine VRML 1.0 and 2.0 ideas of children nodes, but they are generally not used except in some small pieces of code where they just make the code shorter.

```
}
```

Figure 2.1. Four spheres in mixed VRML 1.0 and 2.0 code



This file uses VRML 2.0 sphere that is transformed using both VRML 1.0 approach (`Translation` node that affects all subsequent nodes) and VRML 2.0 approach (`Transform` node that affects all it's children). And everything works: VRML 1.0 nodes are handled according to VRML 1.0 specification, VRML 2.0 according to VRML 2.0 specification. Transformations, no matter which VRML version was used to specify them, affect all shapes. The file's header line says that it's supposed to be VRML 2.0, and this means that when we have a node name that is possible in both VRML specifications (but must be handled differently in each version), for example `Transform` node, file header decides which version of this node (`TNodeTransform_1` or `TNodeTransform_2`) is created when parsing this file.

This also means that you have many VRML 2.0 features available in VRML 1.0. VRML 2.0 nodes like `Background`, `Fog` and many others, that express features not available at all in standard VRML 1.0, may be freely placed inside VRML 1.0 models when using our engine.

Also including (using `WWWInline` or `Inline` nodes) VRML 1.0 files within VRML 2.0 files (and the other way around) is possible. Each VRML file will be parsed taking into account it's own header line, and then included content is actually placed as a children node of including `WWWInline` or `Inline` node. So you get VRML graph hierarchy with nodes mixed from both VRML versions.

2.3. Reading VRML files

You can create a node using `CreateParse` constructor to parse the node. Or you can initialize node contents by parsing it using `Parse` method. However, these both approaches require you to first prepare appropriate `TVRMLLexer` instance and a list of read node names.

There are comfortable routines like `ParseVRMLFile` that take care of this for you. They create appropriate lexer, and may create also suitable `TStream` instance to read given file content.

Some details about parsing:

- Our VRML lexer is a unified lexer for both VRML 1.0 and 2.0. Most of the VRML 1.0 and 2.0 lexical syntax is identical, minor differences can be handled correctly by a lexer because it always knows VRML header line of the given file. So it knows what syntax to expect.

- Note that VRML version of the file from where the node was read is not saved anywhere in the `TVRMLNode` instance. This information is lost after parsing has ended and lexer is destroyed. This is intentional.

  Only while parsing, `ForVRMLVersion` method mentioned earlier may be used to decide which node classes to create based on VRML version indicated in the file's header line.

  This creates a question when saving VRML nodes back to file: what VRML header to write ? We will solve it by `SuggestedVRMLVersion` method, described in Section 2.4.2, “Determining VRML version when writing”.

- To properly handle DEF / USE mechanism we keep a list of known node names while parsing. *After* a node with `DEF` clause is parsed we add the node name and it's reference to `NodeNameBinding` list that is passed through all parse routines. When a `USE` clause is encountered, we just search this list for appropriate node name.

  Simple VRML rules of `DEF` / `USE` behavior make this approach correct. Remember that VRML name scope is not modeled after normal programming languages, where name scope of an identifier is usually limited to the structure (function, class, etc.) where this identifier is declared. In VRML, name scope always spans to the end of whole VRML file (or to the next `DEF` occurrence with the same name, that overrides previous name). Also, the name scope is always limited to the current file — for example, you cannot use names defined in other VRML files (that you included by `Inline` nodes, or that include you)[3].

  The simple trick with adding our name to `NodeNameBinding` *after* the node is fully parsed prevents creating loops in our graph, in case supplied VRML file is invalid.

2.4. Writing VRML files

`SaveToStream` method of `TVRMLNode` class allows you to save node contents (including children nodes) to any stream. Just like for reading, there are also more comfortable routines for writing called `SaveToVRMLFile`.

2.4.1. DEF / USE mechanism when writing

When writing we also keep track of all node names defined to make use of DEF / USE mechanism. If we want to write a named node, we first check `NodeNameBinding` list whether the same name with the same node was already written to file. If yes, then we can place a `USE` statement, otherwise we have to actually write the node's contents and add given node to `NodeNameBinding` list.

The advantages of above `NodeNameBinding` approach is that it always works correctly. Even for node graphs created by code (as opposed to node graphs read earlier from VRML file). If node graph was obtained by reading VRML file, then the DEF / USE statements will be correctly written back to the file, so there will not be any unnecessary size bloat. But note that in some cases if you created your node graph by code then some node con-

[3] Prototypes and external prototypes in VRML 2.0 are designed to handle such non-local node naming.

tents may be output more than once in the file:

1. First of all, that's because we can use DEF / USE mechanism only for nodes that are named. For unnamed nodes, we will have to write them in expanded form every time. Even if they were nicely shared in node graph.

2. Second of all, VRML name scope is weak and if you use the same node name twice, then you may force our writing algorithm to write node in expanded form more than once (because you "overridden" node name between the first `DEF` clause and the potential place for corresponding `USE` clause).

So if you process VRML nodes graph by code and you want to maximize the chances that DEF / USE mechanism will be used while writing as much as it can, then you should always name your nodes, and name them uniquely.

It's not hard to design a general approach that will always automatically make your names unique. VRML 97 annotated specification suggests adding to the node name an _ (underscore) character followed by some integer for this purpose. For example, in our engine you can enumerate all nodes (use `EnumerateNode` method), and for each node that is used more than once (you can check it easily: such node will have `ParentNodesCount + ParentFieldsCount > 1`) you can append `'_' + PtrUInt(Pointer(Node))` to the node name. The only problem with this approach (and the reason why it's not done automatically) is that you will have to strip these suffixes later, if you will read this file back (assuming that you want to get the same node names). This can be easily done (just remove everything following the last underscore in the names of multiply instantiated nodes). But then if you load the created VRML file into some other VRML browser, you may see these internal suffixes anyway. That's why my decision was that by default such behavior is not done. So the generated VRML file will always have exactly the same node names as you specified.

2.4.2. Determining VRML version when writing

Since our engine supports various VRML versions, there appears a question when writing VRML files: what VRML version to indicate in created file header ? One solution would be to save for this purpose in `TVRMLNode` version numbers of the original VRML file from where the node was read. But our engine must also allow easy construction of VRML files by code, so not every node is obtained from parsing some file. Adding some public fields to `TVRMLNode` or parameters for `SaveToVRMLFile` to explicitly indicate desired version is one simple solution, but it's tiresome — it's another piece of information that will have to be figured out and provided when constructing VRML files by code.

Finally the approach we take in our engine places the burden on implementation of each `TVRMLNode` descendant. If you only create nodes of already defined classes, everything should just magically work. Every `TVRMLNode` descendant can override `SuggestedVRMLVersion` method to indicate it's desired VRML version, and how strong is the preference (`SuggestionPriority` parameter). The idea is that node decides about the desired VRML version by collecting desired VRML version of all it's children and then adding his own preference. And there is a `SuggestionPriority` parameter, so that VRML files that use our engine extension that allows to mix VRML 1.0 and 2.0 constructions (see Section 2.2, "The sum of VRML 1.0 and 2.0") will also be written correctly, i.e. using the closest VRML version for them. When writing VRML file, the `SuggestedVRMLVersion` method of root node is called and used to determine header line for the VRML file.

This means that if you have VRML nodes graph (either read from a file or constructed by

code) using only VRML 1.0 features then it will be correctly written as VRML 1.0 file. Same for a file using only VRML 2.0 features. If you have a mixed file that is mostly VRML 1.0 and uses only some VRML 2.0 nodes that easily "match" into VRML 1.0 [4] then the written file will have VRML 1.0 header. Even though it will not be correct standard VRML 1.0 file — it's only correct for our engine. Of course for some ambiguous node graphs there will be no way to determine correct VRML version: for example if you used `IndexedFaceSet` nodes both in VRML 1.0 and 2.0 versions then the generated VRML version header is undefined (actually, it will depend on node order in the graph). This is one real problem of our "sum of VRML 1.0 and 2.0 approach": VRML scene graph in memory is not necessarily valid for either VRML 1.0 or VRML 2.0.

2.4.3. VRML graph preserving

As was mentioned a couple of times earlier, we do everything to get the VRML scene graph in memory in exactly the same form as was recorded in VRML file, and when writing the resulting VRML file also directly corresponds (including DEF / USE mechanism and node names) to VRML graph in memory.

Actually, there are two exceptions:

1. `Inline` nodes load their referenced content as their children
2. When reading VRML file with multiple root nodes, we wrap them in additional `Group` node

... but we work around these two exceptions when writing VRML files. This means that reading the scene graph from file and then writing it back produces the file with the exact same VRML content. But whitespaces (including comments) are removed, when writing we reformat everything to look nice. So you can simply read and then write back VRML file to get a simple VRML pretty-printer.

2.5. Constructing and processing VRML graph by code

This feature was mentioned a couple of times before. In code, you can simply instantiate any nodes you want, you can add them as a children of other nodes, you can set their fields as you like, and so on. Also several methods for enumerating and searching the nodes graph are provided (like `EnumerateNodes` and `FindNode`). See units documentation [http://camelot.homedns.org/~michalis/sources_docs.php] for details.

I made a decent converter from 3DS and Wavefront file formats to VRML 1.0 this way. Once I was able to read these files, it was trivial to construct according VRML graphs for them. You can then save constructed VRML graph to a file (so user can actually use this converter) and you can further process and render them just like any other VRML nodes graph (so my engine seamlessly handles 3DS and Wavefront files too, even though it's almost solely oriented on VRML).

[4] Examples of "easily matching in VRML 1.0" nodes from VRML 2.0 are `Background` and `Fog` nodes. They don't use any VRML 2.0 features like `SFNode` or `MFNode`, and they don't interact with other nodes state in complex way.

Converter to VRML 2.0 using the same approach is also trivial and will be done very shortly.

This also allows VRML authors to include 3DS and Wavefront files inside VRML files by `Inline` nodes, making it possible to handle scenes designed in mixed 3D files formats.

2.6. Traversing VRML graph

Traversing VRML graph means visiting all active VRML graph nodes in a depth-first search order. By “active” nodes we mean that only the visible (or affecting the visible) parts of the graph are browsed — for example, only one child of a `Switch` and `LOD` nodes is visited.

You can traverse nodes using `Traverse` or `TraverseFromDefaultState` methods. For each visited node, a callback function will be called.

The most important feature of traversing is that whole VRML state that we talked about in Section 1.5, “VRML 1.0 state” is collected along the way. For each visited node traverse callback gets all the information about accumulated transformation, active light nodes and (meaningful only for VRML 1.0 nodes) currently bound property nodes (material, texture etc.).

2.7. Shape nodes features

An important descendant of `TVRMLNode` is the `TNodeGeneralShape` class. This is an abstract class. All visible VRML nodes (in VRML 1.0 and 2.0) are descendants of this class. Don't confuse this with `TNodeShape` class — `TNodeShape` represents a VRML 2.0 `Shape` node.

`TNodeGeneralShape` class defines a couple of important methods, overridden in each descendant. All of these methods take a `State` parameter that describes VRML state at given point of the graph (this is typically obtained by a traverse callback), since we need this to have full knowledge about node's geometry.

2.7.1. Bounding boxes

`LocalBoundingBox` and `BoundingBox` methods calculate axis-aligned bounding box of given node.

Axis-aligned bounding box is one of the simplest bounding volume types. It's a cuboid with axes aligned to base coordinate system X, Y and Z axes. It can be easily expressed as a pair of 3D points. In our engine we require that the points' coordinates are correctly ordered, i.e. X position of the first point must always be less or equal than the X position of the second point, and analogously for Y and Z values. We also have the special value for designating empty bounding box. And while we're talking about empty bounding boxes, remember to not confuse empty box with a box with zero volume: a box with zero volume still has some position. For example, a `PointSet` VRML node with only one point has a non-empty bounding box with a zero volume. A `PointSet` without any points has empty bounding box.

I chose axis-aligned bounding boxes just because they are very simple to calculate and operate on. They have some disadvantages — as with all bounding volumes, there is some compromise between how accurately they describe bounding volumes and how comfortable it is to operate on them. But in practice they just work fast and are enough accurate.

`LocalBoundingBox` method returns a bounding box of given object without transforming it (i.e. assuming that `State` contains an identity transformation). `BoundingBox` method takes current transformation into account. Each descendant has to override at least one of these methods. If you override only `LocalBoundingBox` then `BoundingBox` will be calculated by transforming `LocalBoundingBox` (which can give poor bounding volume, much larger than necessary). If you override only `BoundingBox` then `LocalBoundingBox` will be calculated by calling `BoundingBox` with transformation matrix set to identity matrix (this can make `LocalBoundingBox` implementation much slower than a potential special `LocalBoundingBox` implementation that knows that there is no transformation, so no matrix multiplications have to be done).

2.7.2. Triangulating

`VerticesCount` and `TrianglesCount` calculate triangles and vertices count of given shape.

`Triangulate` method actually calculates all the triangles needed to represent given geometry. Simple lines and points are ignored by this method, so you can't use it to render VRML nodes like `PointSet` and `IndexedLineSet`.

These methods take `OverTriangulate` parameter which requires some explanation.

When using Gouraud shading (and that is the case when rendering models in OpenGL) it's desirable to triangulate every large surface — *even if it doesn't improve the geometry approximation by triangles*. This is the inherent problem of Gouraud shading, that says that lighting calculations are done only at the vertices and within the triangles color is interpolated between vertices. This is much faster than calculating light for every pixel, but it also produces inaccurate rendering results when there is “something interesting going on with the lighting” within the triangle. For example when a thin spot light shines at the middle of the triangle, or when the bright specular highlight should appear in the middle of the triangle. Gouraud shading may miss such effects, because the triangle will be completely dark if all three of it's vertices are determined to be dark.

The solution to this is just to avoid problematic situations by using smaller triangles generating more vertices.

For example: when we triangulate quadrics like cylinder and cone, we always approximate circles at their bases as a set of lines, so their side faces are split to many triangles. I call this triangulation “dividing into slices”, after OpenGL documentation, because this triangulation looks like dividing a pizza into slices (when looking from the top of the quadric). But this produces large (tall) triangles that start at the base circle and end at the top (top peak of the cone, or top circle of the cylinder). This is undesirable for Gouraud shading, so we do additional triangulation: we divide cone and cylinder into stacks (like stacks of the tower). Dividing into stacks doesn't improve the quality of our approximation (when we triangulate cone or cylinder, we always only approximate it's real shape), but it helps the shapes to look good when rendering with Gouraud shading.

I call this additional triangulation an *over-triangulation*. While it's useful for Gouraud shading, for many other purposes it's unnecessary and slows down processing (since it creates unnecessarily many triangles). These purposes include collision detection and rendering the scene with other methods, like Phong shading or ray-tracing algorithms.

That's why my triangulation methods allow you to turn this feature on and off as desired by `OverTriangulate` parameter.

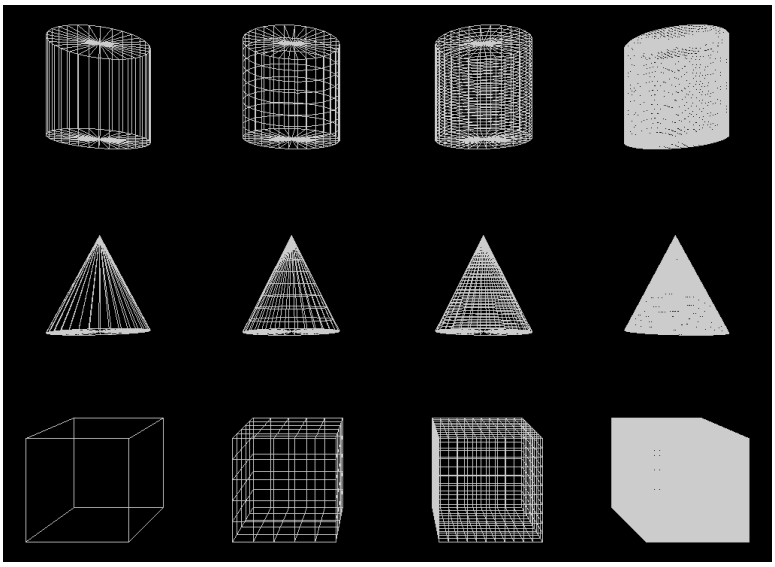
Let's take a look at the following example:

```
#VRML V1.0 ascii

Group {
  PerspectiveCamera {
    position 6 4 14
  }

  PointLight {
    color 0.3 0.3 0.3
    location 6 4 10
  }

  Switch {
    DEF ALight SpotLight {
      location 0 0 3
      direction 0 0 -1
      cutOffAngle 0.3
      color 1 1 0
    }
    DEF Col Separator {
      Separator { USE ALight Cube     { } }
      Translation { translation 0 4 0 }
      Separator { USE ALight Cone     { } }
      Translation { translation 0 4 0 }
      Separator { USE ALight Cylinder { } }
    }
  }

  KambiTriangulation {
    quadricStacks 1 rectDivisions 0 }
  USE Col

  Translation { translation 4 0 0 }
  KambiTriangulation {
    quadricStacks 8 rectDivisions 4 }
  USE Col

  Translation { translation 4 0 0 }
  KambiTriangulation {
    quadricSlices 30 quadricStacks 30 rectDivisions 10 }
  USE Col

  Translation { translation 4 0 0 }
  KambiTriangulation {
    quadricSlices 100 quadricStacks 100 rectDivisions 100 }
  USE Col
}
```

The example shows the cone, the cylinder and the cube with various triangulation. Left-most column has no over-triangulation, next columns have more and more over-triangulation. The VRML code uses the `KambiTriangulation` node to control the triangulation. This node is my VRML extension (hence the prefix `Kambi`), see the page http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_extensions. php#ext_kambi_triangulation for details.

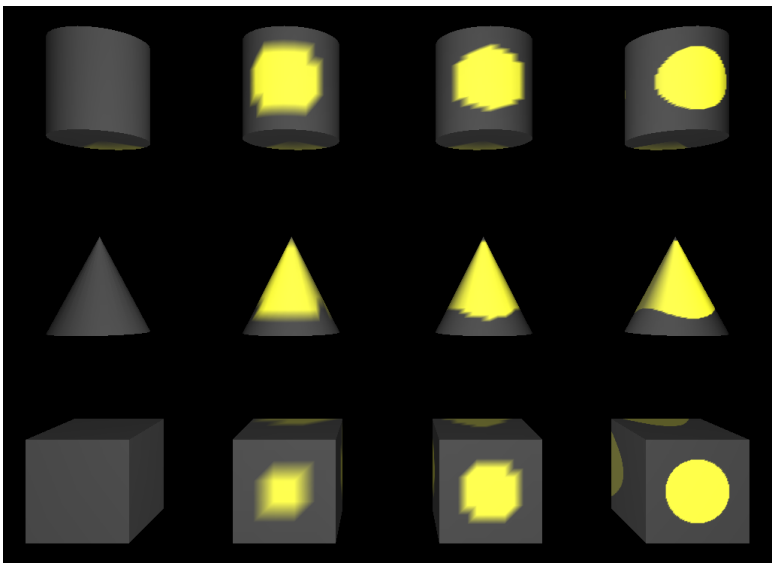
Spot lights shine on every object. First screenshot shows the wireframe view, so you can see how the triangulation looks like.

Figure 2.2. Different triangulations example (wireframe view)



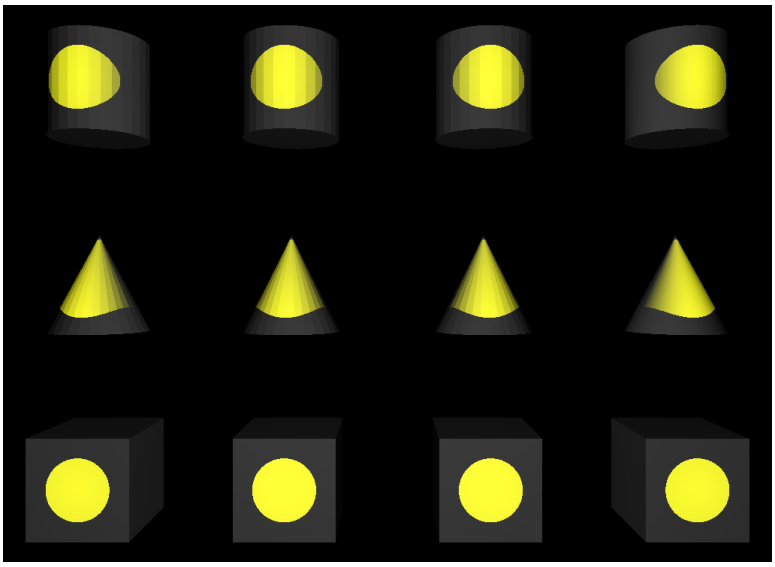
Now let's see the same example rendered using OpenGL (Gouraud shading). As you can see, on the leftmost column spot highlight is not visible at all. The more to the right and the spot looks better and better.

Figure 2.3. Different triangulations example (Gouraud shading)



And finally let's see ray-tracer rendering of the same example. As you can see, over-triangulation (on boxes faces, and stacks on cones and cylinders) doesn't matter here at all.

Figure 2.4. Different triangulations example (ray-tracer rendering)



If you want to control how detailed the triangulation should be:

- Programmers can use `Detail_QuadricSlices`, `Detail_QuadricStacks` and `Detail_RectDivisions` global variables.
- VRML authors can use the KambiTriangulation [http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_extensions. php#ext_kambi_triangulation] VRML node to control this.
- Finally, my programs view3dscene [http:// www. camelot. homedns. org/ ~michalis/ view3dscene.php] and rayhunter [http://www.camelot.homedns.org/~michalis/rayhunter. php] allow you to control this by command-line options

```
--detail-quadric-slices <integer>
--detail-quadric-stacks <integer>
--detail-rect-divisions <integer>
```

2.8. WWWBasePath property

This is a string property that specifies base URL of each node. Actually, for now our engine doesn't support downloading data using any network protocol, so this is always treated just like an absolute path on local file-system. It is always set to the directory of VRML file from which given node was read. It's used by nodes that reference any external file, like `Inline` or `ImageTexture`. Thanks to this field, all such nodes can always resolve their `url` fields with respect to the directory of their file.

For example, assume that inside some directory you have a main VRML file `main.wrl` and two subdirectories: `textures` and `inline`. Inside `textures` you have a file `my_texture.png` and inside `inline` you have VRML file `textured_box.wrl`. Finally, let's say that you want to include textured box in `main.wrl` file, so you write

```
Inline { url "inline/textured_box.wrl" }
```

Now inside `textured_box.wrl` you should reference the texture like

```
ImageTexture { url "../textures/texture.png" }
```

and everything will work when you open `main.wrl` VRML file. Moreover, `textured_box.wrl` is able to "stand on it's own" too, which means that you can open only `textured_box.wrl` and texture will still be properly read.

This is similar to xml:base [http://www.w3.org/TR/xmlbase/] attribute in XML, that was needed to make including XML files by XInclude and referencing external files from various elements (like DocBook's `imagedata`) to cooperate seamlessly.

2.9. Defining your own VRML nodes

At the end it's worth noting that you're not limited to the nodes defined by VRML specifications and implemented in `VRMLNodes` unit. You can freely define your own `TVRMLNode` descendants. All it takes to make them visible is to register them in `NodesManager` object. For example, call

```
NodesManager.RegisterNodeClasses([TNodeMy]);
```

from your unit's initialization section. You may also want to add it to the `AllowedChildrenNodes` list.

This way you can define specific VRML nodes for a specific programs, without the need to modify anything within the base units. I used this technique in the malfunction game [http:// camelot. homedns. org/ ~michalis/ malfunction. php] to define special-purpose VRML nodes like `MalfunctionLevelInfo` and `MalfunctionNotMovingEnemy`.

2.10. Flat scene

If you want to operate on the VRML graph, for some purposes it's enough to load your scene to a `TVRMLNode` instance. This way you know the root node of the scene. Each node points (within it's `Children` property and `SFNode` and `MFNode` fields) to it's children nodes, so if you know the root node of the scene, you know the whole scene. `TVRMLNode` class gives you many methods to operate on the scene graph, and sometimes this is all you need.

However, some operations cannot be implemented in `TVRMLNode` class. The basic reason is that the node doesn't "know" the state of VRML graph where it is used. Node's state is affected by other nodes that may be it's parents or siblings. Moreover, a node may be used many times in the same scene (by DEF / USE mechanism), so it may occur many times in a scene with different states. That's why many `TVRMLNode` methods (like `Triangulate` and `BoundingBox` methods described in Section 2.7, "Shape nodes features") require a parameter `State`: they are not able to figure it out automatically.

These are the reasons why an additional class, called `TVRMLFlatScene`, was created. It is essentially just a wrapper around a VRML root node (kept inside it's `RootNode` property) adding a lot of useful and comfortable methods to operate and investigate the scene as a whole.

2.10.1. List of shape+state pairs

This is the main property of `TVRMLFlatScene`, and the reason for it's name. The idea is simple: to overcome the problems with VRML state, we can just use `Traverse` method from the root node (see Section 2.6, "Traversing VRML graph") and store every shape node (descendant of `TNodeGeneralShape`, see Section 2.7, "Shape nodes features") along with it's state. As a result we get a simple list of pairs: shape and it's state. This list is, to some extent, an alternative "flattened" representation of the VRML graph. Hence the name `TVRMLFlatScene`.

This way we solve various problems mentioned in Section 1.5, “VRML 1.0 state”: we get full accumulated VRML state readily available for each shape. Also we can pick our shape+state pairs in any order, and we can pick any of them. This is crucial for various OpenGL rendering features and optimizations.

For VRML 2.0, some flat scene features were already available. That's because of smart definitions of `children` fields of grouping nodes, as explained earlier in Section 1.5.1, “Why VRML 2.0 is better”: we don't need so much state information in VRML 2.0 and we can pick children of grouping nodes in any order. Still, our flat scene provides the more complete solution: it includes also accumulated transformation matrix and “global” properties (fog and active lights).

Additional advantage of looking at flat scene shapes+state pairs is that resources completely not used (for example `Texture2` node not used by any node in VRML 1.0) are not present. They don't occur in a state of any shape. So unused textures will not be even loaded from their files.

Finally, remember that in Section 1.5, “VRML 1.0 state” we mentioned a practical problem of simple VRML 1.0 implementation in OpenGL: OpenGL stack sizes are limited. Flat scene solves this, because there is no unlimited push/pop hierarchy anymore. Features of nodes like VRML 1.0 `Separator` and `TransformSeparator` are already handled at this stage. And they are handled without using any OpenGL stacks, since this unit can't even depend on OpenGL. Features of VRML 2.0 `Transform` nodes that apply transformation to all it's children are already handled here too.

2.10.2. Various comfortable routines

Numerous other features are available in flat scene:

- Methods to calculate bounding box, vertices count and triangles count of the whole scene. They work simply by summing appropriate results of all shape+state pairs.

- Methods to calculate triangles list (triangulate all shapes in the scene) and to build octrees for the scene. There are also comfortable properties to store the build octree associated with given scene — although our engine doesn't limit how you manage the constructed octrees, you can create as many octrees for given scene as you want and store them where you want.

  More about octrees in Chapter 3, *Octrees*.

- Methods to find `Viewpoint` or camera nodes, transform them, and calculate simple (position, direction, up) triple describing camera setting.

- Methods to find `Fog` node and calculate it's transformation.

- Each shape+state pair is stored as `TVRMLShapeState` instance. This class also has a couple of comfortable routines to calculate bounding box, bounding sphere and such.

2.10.3. Caching

Some flat scene properties are quite time-consuming to calculate. Calculating the list of shape+state pairs requires traversing whole scene graph. Calculating scene bounding box is even more difficult, since for each shape+state we must calculate it's bounding box (in fact calculation of scene bounding box as implemented simply uses the shape+state pairs list).

Obviously we cannot repeat these calculations each time we need these values. So the results are cached inside `TVRMLFlatScene` instance.

Most of the properties are cached: shape+state pairs, bounding boxes, vertices and triangles counts, fog properties. Even triangles' list may be cached if you want.

Also various properties for single shape+state pairs are cached inside `TVRMLShapeState` instance: bounding box, bounding sphere and triangle and vertices counts. After all, some of these operations are quite time-consuming by themselves. For example to calculate bounding box of `IndexedFaceSet` we have to iterate over all it's coordinates.

Changes to actual VRML nodes are not automatically detected. In other words cache is not automatically cleared on changes. Instead you have to manually call appropriate `ChangedXxx` methods of flat scene after changing some parts of the scene. You can specify information about changes in various ways, and as few as possible parts of the cache should get cleared.

For example when you call `ChangedFields(Node)` method then flat scene checks what class your `Node` has and what it can affect. Changes to VRML 1.0 nodes like `Texture2` or `Material` will affect only the shape+state pairs that have these nodes in their state. So the whole shape+state list doesn't need to be regenerated, also the information about other shape+state pairs (like their bounding boxes) is still valid.

In Section 5.3, “Flat scene for OpenGL” we will introduce the `TVRMLFlatSceneGL` class that descends from `TVRMLFlatScene`. It adds various OpenGL methods and caches various OpenGL resources related to rendering particular scene parts. This means that our `ChangedXxx` methods will have even greater impact.

Chapter 3. Octrees

Octree is a tree structure used to partition a 3D space. Each octree node has eight children (hence the name “octree”, oct + tree). Our engine uses octrees for a couple of tasks.

3.1. Collision detection

Generally speaking, octree is useful for various collision detection tasks:

1. First of all, for a “normal” collision detection needed in games. That is for checking collisions between the player and the world geometry. The player may be represented by a sphere, and when the player moves we check that:

   - The line segment between the current player position and the new player position does not collide with the world.
   - The sphere surrounding new player position does not collide with the world.

   When we detect a collision, we can simply reject player move, or (much better) propose another, non-colliding new player position. This way the player can “slide” along the wall when he tries to move into it.

   This is done within `MoveAllowed` and `MoveAllowedSimple` methods of `TVRMLTriangleOctree` class.

   Also, when gravity works, we want the player to preserve some preferred height above the ground. This allows the player to climb up and down the hills, stairs etc. It is often called *terrain-following*. This requires calculating current player height above the ground. By comparing this height with a preferred height we know whether the player position should fall down or raise up. This is done by checking for a collision between a ray (that starts at player's position and is directed down) with the world.

   This is done by `GetCameraHeight` method of `TVRMLTriangleOctree` class.

2. For ray-tracer, this is the most important data structure. Ray-tracer checks collisions of rays with the world to calculate it's image. Also when calculating shadows we check for collision between light point (or a random point on light's surface, in case of surface lights) and the possibly shadowed geometry point.

   This is done by `RayCollision` and `SegmentCollision` methods of `TVRMLTriangleOctree` class.

3. When player picks (for example by clicking with mouse) given point on the screen showing 3D scene, we want to know which object from our 3D scene (for example, which VRML node) he actually picked. So again we want to do collision detection between a ray (starting at player's position and with direction calculated from player's looking direction, screen dimensions and picked point coordinates on the screen) and the world.

   Note that there are other methods to determine which object player picked. For example you could employ some OpenGL tricks: rendering in selection mode, or reading color buffer contents to get results of depth buffer tests. See The OpenGL Programming Guide - The Redbook [http://www.opengl.org/documentation/red_book/] for details. But once we have octree already implemented, it is usually easier and less cumbersome

to use than these tricks.

4. When rendering using OpenGL, we don't want to pass to OpenGL objects that are known to be invisible to the player. For example, we know that objects outside of the camera frustum are invisible. In certain cases (when e.g. dense fog is used) we also know that objects further from player than certain distance are not visible.

   This means that we want to check for collision between camera frustum and/or sphere with the world. This is done by `EnumerateCollidingOctreeItems` and `SphereCollision` methods.

   More information about how these algorithms are used will be given in Section 5.3, “Flat scene for OpenGL”.

3.2. How octree works

Octree is a tree where each internal (non-leaf) node has eight children. Each node spans a particular space area, expressed as an axis-aligned bounding box (available as `Box` property of `TOctreeNode`). Each node also has a chosen middle point inside this box (available as `MiddlePoint` property of `TOctreeNode` class). This point defines three planes parallel to the base X, Y and Z planes and crossing this point. Each child of given octree node represents one of the eight space parts that are created by dividing parent space using these three planes.

Each child, in turn, may be either

1. Another internal node. So it has his own middle point and another eight children. His middle point must be within the space part that his parent node gave him.

2. Or a leaf, that simply contains actual items that you wanted to store in an octree. What is an “actual item” depends on with want items you want to calculate collisions using this octree.

   In our engine we have two octree types:

   a. `TVRMLTriangleOctree` that keeps triangles

   b. `TVRMLShapeStateOctree` that keeps VRML nodes of `TNodeGeneralShape` class (remember from Section 2.7, “Shape nodes features” that these are the only VRML nodes that actually have some geometry visible) along with their `State` (obtained from traversing VRML graph).

What happens when given item should be included in more than one children ? That is, item is contained in space part of more than one children ?

1. Simple solution is to put this item inside all children where it should be. This means that we could waste a lot of memory if given item should be present in many leaf nodes, but this problem can be somewhat cured by just keeping an array of octree items for the whole octree (`OctreeItems` property of each octree class) and keeping only indices to this array in octree leafs (`ItemsIndices` property of `TOctreeNode`).

2. Another possible solution is to keep such problematic item only in the list of items of internal node, instead of putting it inside children nodes. But each octree node has eight

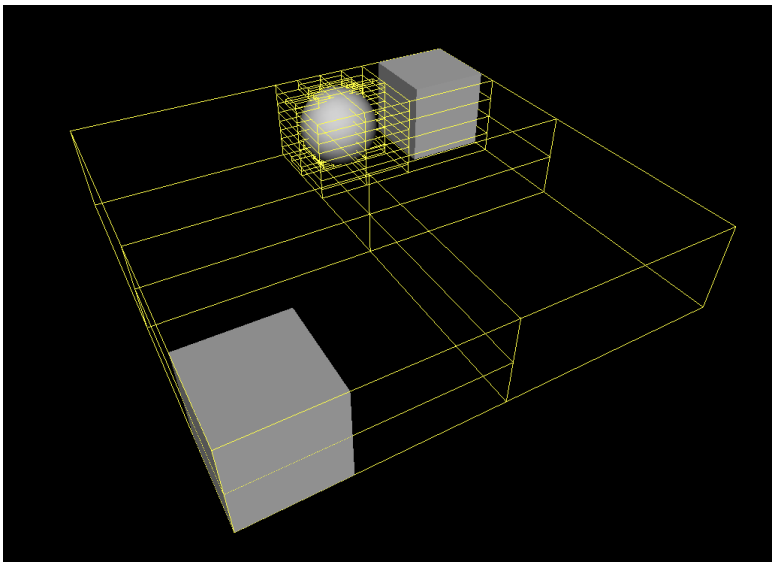
children, and given item can be contained for example only in two of eight children. In this case our collision checking routines would always have to consider this item, while in fact they should consider it only for a 2/8 part of the space.

That's why my engine doesn't use this approach. Note that some hybrid approach could be possible here, for example keep the item if it spans more than 4 children nodes and put it inside children otherwise. This idea remains to be implemented one day... For now our collision checking is fast enough for all purposes when it's needed in real-time games.

Example below shows an octree constructed by our engine. The sample scene contains two boxes and a sphere. On the screenshot yellow bounding boxes indicate every internal node and every non-empty leaf. Whole scene is contained within root node of the tree, so the largest yellow bounding box corresponds also to the bounding box of the scene. The “lonely” box (in the foreground) is placed within the two direct children on the root tree node. Left and right quarter on the image contain only empty children leaves of root node, so their bounding boxes are not shown. Finally, the interesting things happen in the quarter with a box and a sphere. Sphere has many triangles, so a detailed octree is constructed around it. Also the sphere caused a little more detailed octree around the near box.

```
#VRML V2.0 utf8

Viewpoint  {
  position -10.642 8.193 -5.614
  orientation -0.195 -0.921 -0.336 2.158
}

Transform {
  translation 4 0 1.25
  children Shape {
    appearance DEF ALit Appearance { material Material { } }
    geometry Sphere { }
  }
}

Transform {
  translation 4 0 4
  children Shape {
    appearance USE ALit
    geometry Box { }
  }
}

Transform {
  translation -4 0 -4
  children Shape {
    appearance USE ALit
    geometry Box { }
  }
}
```

Figure 3.1. A sample octree constructed for a scene with two boxes and a sphere



You can view octree like this using view3dscene [http:// www. camelot. homedns. org/ ~michalis/ view3dscene. php]. Just turn on the menu option “View” -> “Show whole octree”. There are also menu commands to investigate octree nodes only at the particular depth.

3.2.1. Checking for collisions using the octree

Let's assume that you have some reference object (like a sphere or a ray or a line segment mentioned in the first section) that you want to check for collisions with all items contained in the octree. You start from the root node — all items, which means “all potential colliders”, are there. You check with which children of this node your object could possibly collide. Different object types will require various approaches here. In general, this comes down to checking for collision between children nodes' boxes and your reference object. For example:

1. For a sphere, you check which child node contains the sphere center. Then you check with which planes (of the three dividing planes of this node) the sphere collides. This determines all the children that the sphere can collide with.

   Above approach is not as accurate as it could be — since it effectively checks the collision of the bounding box of the sphere with children boxes. To make it more accurate you can check whether the middle point of given node is within the sphere. But it's not certain whether this additional check will make your collision detection faster (because we will descend into less children nodes) or slower (because we spend time on the additional check). In practice, this depends on how large spheres you will check for collision — for small (small in comparison to the world) spheres, this additional check will seldom eliminate any child and probably will be worthless.

2. For a ray: determine child node where ray start is. Then check for collision between this ray and three base planes crossing node's middle point. This will let you determine into which children nodes the ray enters. Similar approach could be taken for the line seg-

ment.

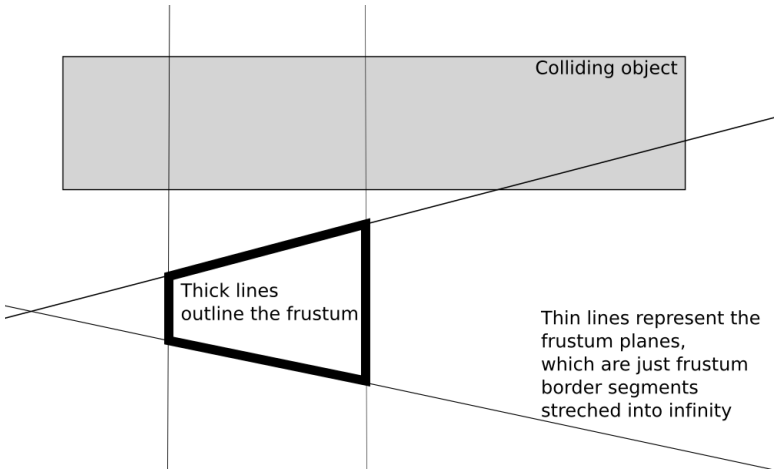
3. For a frustum: first note that our engine stores frustum as a 6 plane equations.

   The basic approach here is to employ the method of checking for collision between a plane and a box. To determine collision of a box with a plane you can check 8 box corners on which side of the plane they are (simply by checking expression similar to the plane equation, *Ax + Bx + Cz + D >= 0*). If all points are on the same side of the plane (and no point lies precisely on the plane) then there is no collision. This also tells your *on which size of the plane* the box is located, in case there is no collision.

   In our engine, frustum planes are correctly oriented, so the answer to the question “on which size of the plane” a box is located is meaningful to us. To check for collision of frustum with a node, we check 6 frustum planes for collision with this node's box. If box is on the *inside side of every plane*, this means that the box is completely inside the frustum. Otherwise, if the box is on the *outside side of at least one plane*, then the box is completely outside of the frustum. Otherwise (which means that box collides with at least one plane, and it's not outside any plane) we don't really know.

   In the last case, we're pretty certain that the box collides somehow with the frustum, so we assume this. In case of error, nothing terrible will happen: our collision checking routine using octree will just work a little slower than possible, but it will still be 100% correct. In practice, in almost all cases our assumption will be true, although some nasty cases are indeed possible. You can see an example of such case below. This is a side view showing a frustum and a box. You can see that the box collides with 3 planes and is considered to be on the inside of the 4th plane (the one at the bottom). You can easily extend this image to 3D and imagine the remaining 2 frustum planes in such way that they will intersect the box.

Figure 3.2. A nasty case when a box is considered to be colliding with a frustum, but in fact it's outside of the frustum



Once you can check with which octree node's children your object collides, you just apply this process recursively. That is, for each internal node you determine which of it's children may collide with your reference object, and recursively check for collisions inside these children. For each leaf node, you just sequentially check all it's items for collision. For example, in case of a triangle octree, in the leafs you will check for collision between tri-

angles and your reference object.

What's the time of this collision checking algorithm ? Like with all tree structures, the idea is that the time should be logarithmic. But actually we don't use any advanced techniques that could ensure that our octree is really balanced. And the fact that items that are put inside more than one children are effectively multiplied in the octree doesn't help either. However octrees of real-world models are enough balanced (and multiplication is small enough) to make collision checking using octrees “logarithmic (i.e. fast) in practice”.

Some more notes about collision checking using an octree:

- Sometimes all you need is the information that “some collision occured” (for example that's enough for shadow detection). Sometimes you want to get the closest collision point (for example, closest to the ray start, for ray-tracing). The first case can obviously be optimized to finish whole algorithm as soon as any collision is found. In the second case you must always check all items when you process a leaf node (because the items in leaf nodes are not ordered in any way). But when processing internal nodes it can still be optimized to not enter some children nodes if collision in earlier child node was found (in cases when we know that every possible collision in one child node must be closer to ray start than every possible collision in other node).

- As was mentioned earlier, if an octree item fits into more than one child of given node, we put it inside every matching children node, thus duplicating information about this item in many leafs. But this means that we can lose some speed. We can be fooled into checking more than once for collision between our reference object (like a ray) with *the same item, but placed within a different leaf*.

  This is not so terribly bad, since we are talking here about tests like checking for collision between a single ray and a single triangle. So this test is anyway quite fast operation, in constant time. But still it requires a couple of floating point operations, and it's called very often by our algorithm, so we want to optimize it.

  The solution is called *the mailboxes*. Each octree item gets a mailbox. Each reference object (like a ray) gets a unique tag. Before we check for collision between our reference object and an octree item, we check whether the mailbox has the information about the collision test result for this object tag. If yes, then we obtain the collision test result from the mailbox. Otherwise, we perform normal (more time-consuming) test and we store the test result along with the object tag within the mailbox. This way each item will be tested for collision with reference object only once. Next time we will just use the mailbox.

  This is possible to implement thanks to the fact that we keep indices to items in octree nodes, and the actual items are kept in an array for a whole octree. So we can naturally place our mailboxes in this array.

3.2.2. Constructing octree

A simple algorithm starts with an empty tree, containing one leaf node with no items. Then we add our items (triangles, VRML shape nodes etc.) to the octree keeping an assertion that no leaf can have more than some specified number of items (`MaxLeafItemsCount` property of `TOctree` class). When we see that adding another item to some leaf would break this assertion, we convert the leaf to an internal node with eight children, and we add items (previous leaf items and the new item that we're trying to add) to newly created children. Of course, each children gets only the items that are within its space part.

Note that this algorithm doesn't guarantee in any way that a tree is balanced. And we want the tree to be balanced, otherwise checking for collisions using this tree will be as slow (or even slower) than just sequentially checking collision with all items. However, for most real-world models, the items are spread more-or-less evenly across the scene, so in practice our tree is more-or-less balanced. To prevent the pathological cases that could result in extremely deep octrees we can add a simple limit on the allowed depth of the tree (`MaxDepth` property of `TOctree` class). When a leaf reaches `MaxDepth`, we will not split it to an internal node anymore, no matter how many items does it contain. So the assertion becomes "leafs on depth < `MaxDepth` must have at most `MaxLeafItemsCount` items". This way the nasty cases are somewhat bounded — our collision checking using tree cannot be *much* slower than just sequentially checking for collision with all contained items.

There is a question how to calculate middle point of each node. The simple and most common approach is just to calculate it as an actual middle point of the node's box. Root tree node gets a box equal to the bounding box of our scene. But you could plug here other techniques. The basic idea is that the tree should be balanced, so ideally the middle point should divide the node's box into eight parts with equal number of triangles inside.

For some purposes it's helpful to keep in each internal node a list of all items contained within it's children. This eats more memory, but may allow in some cases to terminate the collision checking operation faster. For example, when we want to check which octree items are inside a camera frustum, we often find ourselves in a situation when we know that some octree node is completely contained within the frustum. If we have all the items' indices easily accessible within this internal node, we can avoid having to traverse all children nodes under this node. This is used by `TVRMLShapeStateOctree` in our engine.

3.3. Similar data structures

There are other tree structures similar to the octree. Generally, octree is the easiest one to construct. Other tree structures give greater flexibility how the space is partitioned on each level, but to actually get the significant speed benefit, these trees must be also constructed in much smarter way.

- *kd-tree* partitions space at each node by a plane parallel to one of the base planes. In other words, it uses one plane where octree uses three planes. This allows greater flexibility, for example it may be more optimal to divide the space more often by a *X = const* plane than *Y = const*. Octree is forced to divide space by all three planes at each node.

  If you will use the simple "rotational" strategy (X, Y, Z, then again X, Y, Z and so on) to choose partitioning axes at each depth, then the kd-tree becomes similar to an octree.

  The name kd-tree comes from "k-dimensional tree" term, since kd-tree may be used for any number of dimensions, not necessarily 3D.

- *BSP (Binary Space Partitioning) tree* partitions space in each node by a plane. *Any plane*, not necessarily parallel to one of the base X, Y, Z planes.

  This gives even more flexibility than kd-tree, but it makes constructing optimal BSP trees much harder (assuming that you want to actually produce a better tree than what can be achieved with kd-tree). It also means that at each node you have to check for collision between your reference object and a free plane (instead of a plane parallel to one of the base coordinate system planes), so computations get a little slower than for kd-tree.

Note that BSP tree is suitable for any number of dimensions, just like kd-tree. You just use different equations to represent hyperplanes in other dimensions.

- Finally, note that the only thing that ties octree to 3 dimensions is actually it's name. The same approach could be used for any number of dimensions. For N dimensions, each internal node will have 2^N children. For example for 2 dimensions each node has 4 children, and such tree is called a *quadtree*.

  Note that this approach is completely inadequate when we have a really large number of dimensions (because than 2^N will be so large that “organizational” data of all tree nodes may eat a lot of memory), but this is not a problem as long as we stay within 2 or 3 dimensions.

Chapter 4. Ray-tracer rendering

This chapter describes our implementation of ray-tracer, along with some related topics.

We don't even try to explain here how ray-tracing algorithms work, as this is beyond the scope of this document. Moreover, the ray-tracer is not the most important part of our engine right now (OpenGL real-time rendering is). This means that while our ray-tracer has a couple of nice and unique features, admittedly it also lacks some common and important ray-tracer features, and it certainly doesn't even try to compete with many other professional open-source ray-tracing engines existing.

Many practical details related to using our ray-tracer are mentioned in rayhunter documentation [http://www.camelot.homedns.org/~michalis/rayhunter.php]. Many sample images generated by this ray-tracer are available in the rayhunter gallery [http://www.camelot.homedns.org/~michalis/raytr_gallery.php].

4.1. Using octree

The basic data structure for ray-tracing is an octree based on triangles, that is `TVRMLTriangleOctree` instance. If you want to ray-trace a scene, you have to first build such octree and pass it to a procedure that does actual ray-tracing. Note that the quality of the octree is critical to the speed of the ray-tracer. Fast ray-tracer requires much deeper octree, with less items in leafs (`MaxLeafItemsCount` property) than what is usually sufficient for example for collision detection in real-time game.

To calculate triangles for your octree you should use the `Triangulate` method of VRML shape nodes. Triangles enumerated by this method should be inserted into the octree. If you use `TVRMLFlatScene` class to load VRML models (described in Section 2.10, “Flat scene”) you have a comfortable method `CreateTriangleOctree` available that takes care of it all, returning the ready octree for a whole scene.

The `Triangulation` method is also admittedly responsible for some lacks in our ray-tracer. For example, ray-tracer doesn't handle textures, because triangulation callback doesn't return texture coordinates. Also normal vectors are not interpolated because triangulation callback doesn't return normal vectors at the triangle corners. This is all intended to be fixed one day, but for now ray-tracer is not that important for our engine.

4.2. Classic deterministic ray-tracer

Classic Whitted-style deterministic ray-tracer is done by `ClassicRayTracerTo1st` procedure in `VRMLRayTracer` unit.

Point and directional lights are used, as defined by all normal VRML light nodes. This means that only hard shadows are available. Algorithm sends one primary ray for each image pixel. Ray-tracing is recursive, where the ray arrives on some surface we check rays to light sources and eventually we recursively check refracted ray (when `Material` has `transparency` > 0) and reflected ray (when `Material` has `mirror` > 0).

The resulting pixel color is calculated according to VRML 97 lighting equations [http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/part1/concepts.html#4.14]. This is probably the most important advantage of ray-tracer in our engine: ability to calculate images conforming precisely to the VRML 97 lighting specification. Actually, we modified these equations a little, but only because:

1. I have recursive ray-tracing while VRML 97 specifies only local light model, without a placeholder for reflected and refracted color.

2. VRML 1.0 `SpotLight` must be calculated differently, since it uses the `dropOfFRate` field (a cosinus exponent) to specify spot highlight. While VRML 2.0 uses the `beamWidth` field (a constant spot intensity area and then a linear drop to the spot border). So for VRML 1.0 spot lights we use the equations analogous to the OpenGL lighting equations.

3. The ambient factor is calculated taking into account that standard VRML 1.0 light nodes don't have the `ambientIntensity` field. Although, as an extension, our engine allows you to specify `ambientIntensity` to get VRML 2.0 behavior in VRML 1.0 [http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_extensions. php#ext_light_attenuation].

4.3. Path-tracer

Done by `PathTracerTo1st` procedure in `VRMLRayTracer` unit.

Surface lights are used: every shape with non-zero `emissiveColor` is considered a light emitter. For each image pixel many random paths are checked and final pixel color is the average color gathered from all paths.

Path length is determined by a given minimal path length and a Russian-roulette parameter. Every path will have at least the specified minimal length, and then Russian-roulette will be used to terminate the path. E.g. if you set minimal path length to 3 and Russian-roulette parameter to 0.5 then 1/2 of all paths will have length 3, 1/4 of all paths will have length 4, 1/8 of all paths will have length 5 etc.

Russian-roulette makes sure that the result is *unbiased*, i.e. the expected value is the correct result (the perfect beautiful realistic image). However, Russian-roulette introduces also a large variance, visible as a noise on the image. That's why forcing some minimal path length helps. Sensible values for minimal path length are around 1 or 2. Of course, the more the better, but it will also slow down the rendering. You can set minimal length to 0, then Russian-roulette will always be used to decide about path termination (expect a lot of noise on the image!).

Actually our path-tracer does something more than a normal path-tracer should: for every pixel it checks `PrimarySamplesCount` of primary rays, and then each primary ray that hits something splits into `NonPrimarySamplesCount`. So in total we check `PrimarySamplesCount * NonPrimarySamplesCount` paths. This optimization comes from the fact that there is no need to take many `PrimarySamplesCount`, because all primary rays hit more-or-less the same thing, since they have very similar direction.

To get really nice results path-tracer requires a different materials description. I added a couple of additional fields to `Material` node to describe physical material properties (for Phong's BRDF) [http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_extensions. php#ext_material_phong_brdf_fields]. If these fields are not specified in `Material` node, path-tracer tries to calculate them from normal material properties, although this may result in a poor-looking materials. There's also a program kambi_mgf2inv [http://www.camelot. homedns.org/~michalis/kambi_mgf2inv.php] available that let's you convert MGF files to VRML 1.0 generating correct values for these additional `Material` fields.

Shadow cache is used, this makes path-tracer a little faster. Also you can generate the im-

age pixels in more intelligent order than just line-by-line: you can use Hilbert or Peano space-filling curves. In combination with shadow cache this can make path-tracing faster (shadow cache should hit more often). Although in practice space-filling curves don't make any noticeable speed difference. Undoubtedly, there are many possibilities how to improve the speed of our path-tracer, and maybe one day space-filling curves will come to a real use.

4.4. RGBE format

Our ray-tracer can store images in the RGBE format.

RGBE stands for *Red + Green + Blue + Exponent*. It's an image format developed by Greg Ward, and used e.g. by Radiance [http://floyd.lbl.gov/radiance/]. Colors in RGBE images are stored with a very good precision, while not wasting a lot of disk space. *Good precision* means that you may be able to expose in the image some details that were not initially visible for the human eye, e.g. by brightening some areas. Also color components are not clamped to [0; 1] range — each component can be any large number. This means that even if resulting image is too bright, and some areas look just like white stains, you can always correct the image by darkening it or applying gamma correction etc. This is especially important for images generated by path-tracer.

You can process RGBE images using various Radiance programs. You can also use RGBE images in all my programs, for example you can view them using glViewImage [http://www.camelot.homedns.org/~michalis/glviewimage.php] and you can use them as textures on VRML models.

4.5. Generating light maps

This is a feature closely related to ray-tracer routines, although it doesn't actually involve any recursive ray-tracing. The idea comes from the realization that we already have a means to calculate light contribution on a given point in a scene, including checking what lights are blocked on this point. So we can use these methods to calculate lighting on some surface *independent of the camera (player) position*. All it takes is just to remove from lighting equations all components related to camera, which means just removing the specular component of lighting equation. We can do it even for any point on a scene (not necessarily a point that is actually part of any scene geometry), as long as we will provide material properties that should be assumed by calculation.

What do we get by this ? We get the ability to generate textures that contain accumulated effect of all lights shining on given surface. This includes shadows. We can use such texture on a surface to get already precomputed lighting *with shadows*. Of course, the trick will only work as long as lights are static in the scene and it's not a problem to remove specular component for given surface. And remember to make the texture large enough — otherwise user will see that the shadows on the wall are pixelated and the whole nice effect will be gone.

I used this trick to generate ground texture for my toy lets_take_a_walk [http://www.camelot.homedns.org/~michalis/lets_take_a_walk.php]. Initially I had this model:

Figure 4.1. lets_take_a_walk scene, side view

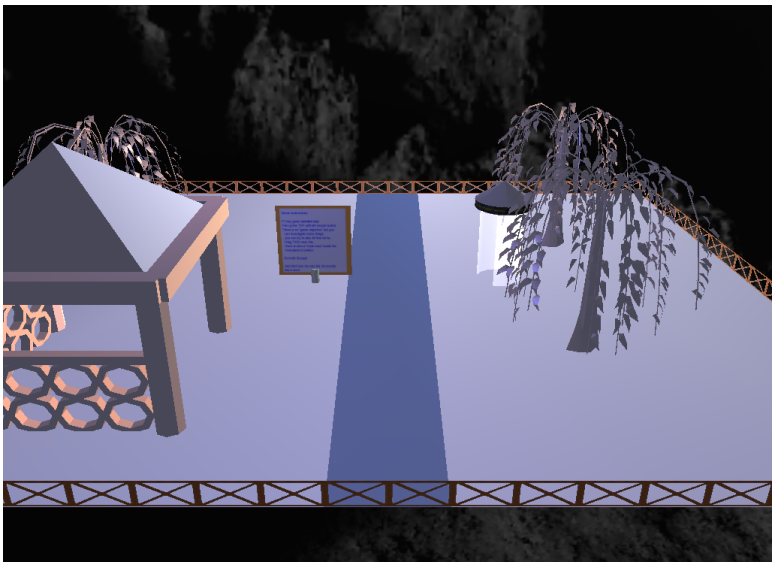
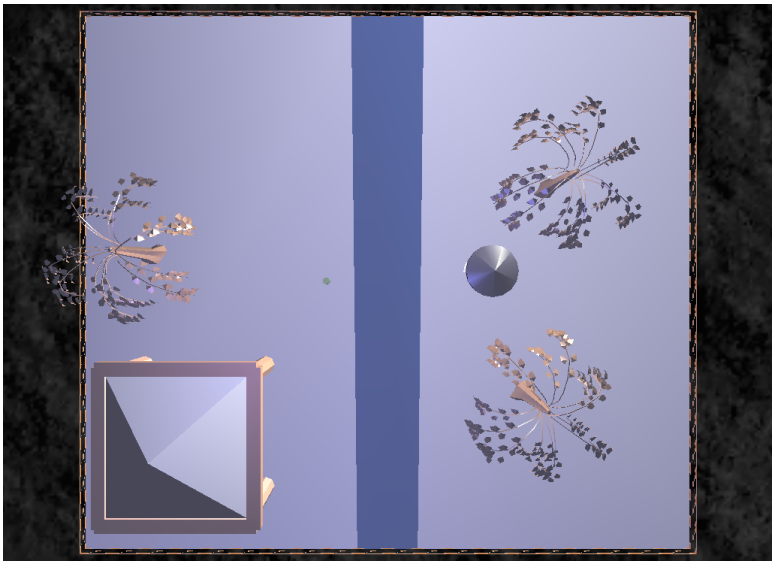
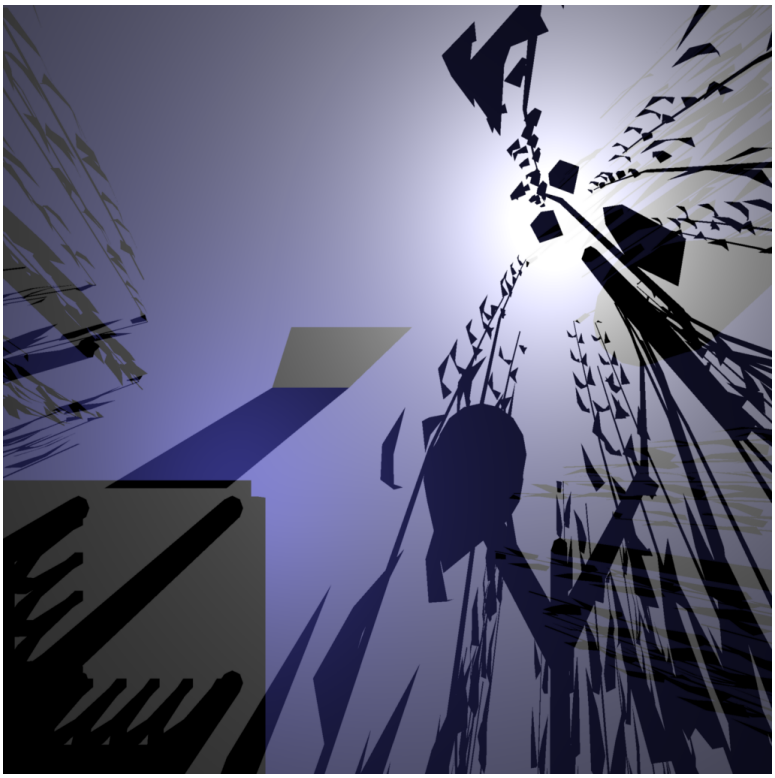


Figure 4.2. lets_take_a_walk scene, top view



Using our trick I generated this texture for the ground. Note how the texture includes shadows of all scene objects. And note how the upper-right part of the texture has a nice brighter area. Our OpenGL rendering above didn't show this brighter place, because the ground geometry is poorly triangulated. So OpenGL rendering hit again the problems with Gouraud shading discussed in detail earlier in Section 2.7.2, “Triangulating”. It's a quite large texture (1024 x 1024 pixels), but any decent OpenGL implementation should be able to handle it without any problems. In case of problems, I would just split it to a couple of smaller pieces.

Figure 4.3. Generated ground texture



Finally, resulting model with a ground texture:

Figure 4.4. lets_take_a_walk scene, with ground texture. Side view

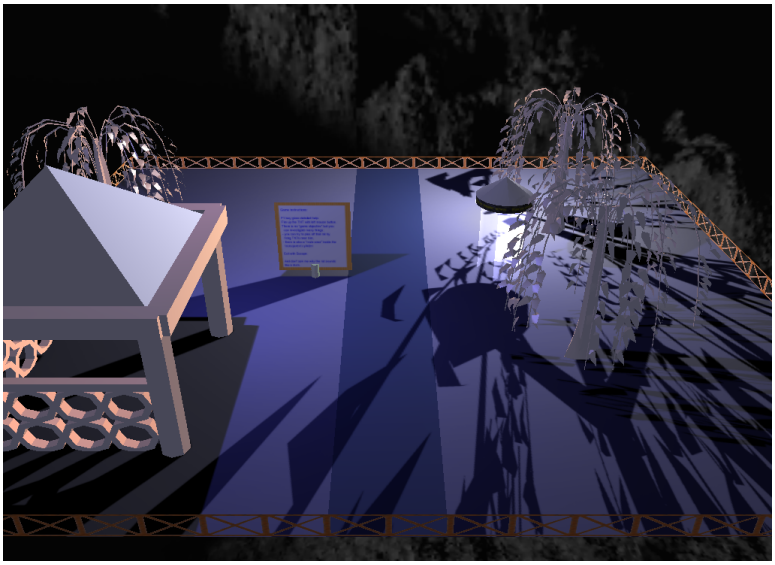
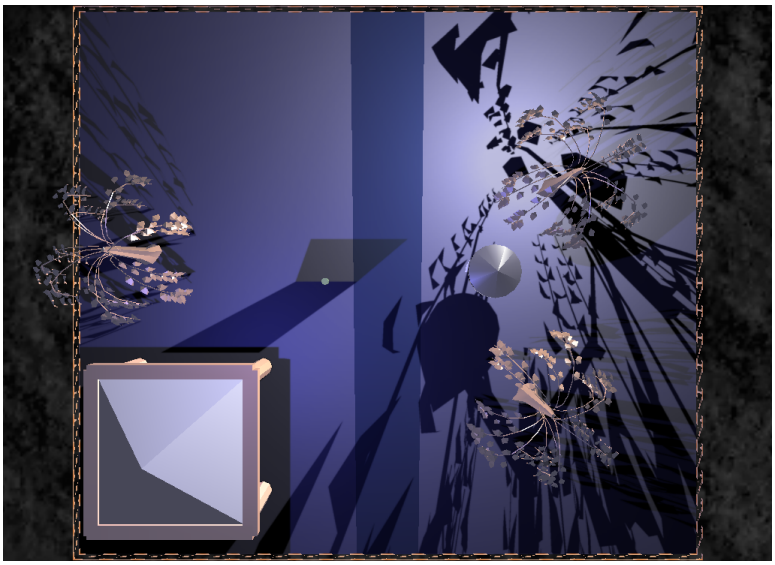


Figure 4.5. lets_take_a_walk scene, with ground texture. Top view.



Such textures may be generated by the `gen_light_map` program included in the `units/3dmodels/tools/gen_light_map.dpr` file in our engine source code. The underlying unit responsible for all actual work is called `VRMLLightMap`. `lets_take_a_walk` source code is available too, so you can see there an example how the `gen_light_map` program may be called.

Chapter 5. OpenGL rendering

5.1. VRML lights rendering

5.1.1. Lighting model

When rendering using the OpenGL we try to get results as close as possible to the VRML 97 lighting equations [http:// www. web3d. org/ x3d/ specifications/ vrml/ ISO-IEC-14772-VRML97/ part1/ concepts. html#4. 14]. We look at equations calculated by OpenGL for lighting, materials, fog etc. and set OpenGL properties such that the OpenGL results should match the results required by VRML 97 specification.

There are cases when it is not possible to match VRML 97 requirements precisely by this approach:

1. VRML 2.0 way of specifying spot light highlight, by `beamWidth`, cannot be translated to a standard OpenGL spotlight.

   Let's look at the equations. Let α be the angle between the spot light's direction and the ray from spot light's position to the considered geometry point.

   - OpenGL spot light uses cosinus drop-off, which means that the light intensity within the spot `cutOffAngle` is calculated as a $Cos(\alpha)^{spotExponent}$.

   - VRML spot light has a constant spot intensity area and then a linear drop-off to the spot border. In other words, the light intensity is 1.0 (maximum) when $\alpha <$ *beamWidth* and it drops linearly from 1.0 to 0.0 as α raises up from `beamWidth` to `cutOffAngle`.

   There is no general translation possible between these two systems, so VRML 2.0 spot light cannot be perfectly represented by a standard OpenGL spot light.

   The good news is that the default VRML case, where `beamWidth` > `cutOffAngle` can be represented correctly in OpenGL (just set the spot exponent to 0.0).

2. The exponential fog of VRML 2.0 also uses different equations than OpenGL exponential fog and cannot be matched perfectly. See VRML and OpenGL specifications for details.

3. Gouraud shading limitations, discussed a couple of times before, are another OpenGL rendering limitation that simply cannot be overcome.

Still, OpenGL rendering is usually quite close to what VRML 97 specification requires. An easy way to check this conformance is to compare the OpenGL renderings with the renderings done by the classic ray-tracer.

5.1.2. Rendering lights separately

VRML lights can be translated to appropriate OpenGL calls using `TVRMLLightSetGL` class. This is normally used internally by `TVRMLOpenGLRenderer` class that will be discussed in next sections, but it can also be used separately for special purposes. For example, in games you may want to render various things in OpenGL context: maybe you

render many VRML models (for example you have one static world VRML model and various creature VRML models) and maybe you render some things straight to OpenGL (i.e. not through our VRML engine). If you want to have complete control over what is affected by VRML lights, you can load them to `TVRMLLightSetGL` instance. Then you can explicitly say when these lights are rendered for OpenGL and so you have complete control over what is lightened by them.

I use this technique in my games. For example see "The Castle" [http://www.camelot.homedns.org/~michalis/castle.php] levels. They use special `xxx_lights.wrl` VRML files that contain only light nodes. You can even edit (and write back as VRML files) these lights at run-time from the game. Use debug menu "Edit light" to do this. This makes a nice educational tool that allows you to experiment with VRML lights properties.

5.2. Basic OpenGL rendering

`TVRMLOpenGLRenderer` class does the basic OpenGL rendering of VRML nodes. "Basic" rendering means that this class is not supposed to

- Choose the order of rendering of VRML nodes. This implicates that `TVRMLOpenGLRenderer` is not responsible for doing optimizations that pick only some subset of VRML nodes for rendering (for example, only the nodes visible within the camera frustum). This also implicates that it's not responsible for arranging the rendering order for OpenGL blending, see Section 5.3.1, "Material transparency using OpenGL alpha blending". In fact, it doesn't set any OpenGL blending parameters (aside from setting colors alpha values as appropriate).

- Wrapping rendering of VRML nodes within OpenGL display lists. `TVRMLOpenGLRenderer` class always renders by "real" calls to OpenGL, no display lists. However, it's carefully designed to allow the caller to wrap most of it's rendering methods inside OpenGL display lists.

Above limitations are done by design. It's supposed that a higher-level routines will internally use instance of this class to perform rendering. These higher-level routines should then provide support for all things mentioned above, like choosing appropriate nodes order and wrapping calls in OpenGL display lists. In the next Section 5.3, "Flat scene for OpenGL" we will get familiar with such higher-level class.

The way to use `TVRMLOpenGLRenderer` looks like this:

1. First you must call `Prepare` method for all the `State` instances that you want to later use for rendering. You can obtain such `State` instances for example by a traverse callback discussed earlier in Section 2.6, "Traversing VRML graph". The order of calling `Prepare` methods doesn't matter — it's only important for you to prepare *all states* before you will render them.

   `Prepare` calls cannot be wrapped inside OpenGL display lists. In fact, the very reason why they exist is that there are some places where `TVRMLOpenGLRenderer` wants to create an internal OpenGL display list that can be used later when rendering.

   Right now `Prepare` calls prepare two kinds of resources: they load textures into OpenGL, and they load outline fonts (used by VRML `Text` and `AsciiText` nodes) into display lists.

   You are free to mix `Prepare` calls with any other rendering calls to OpenGL. This

doesn't matter, as `Prepare` only prepares some resources, without changing OpenGL state. The only thing that you are forbidden to do is to mess (use or delete) display lists and texture names reserved inside `Prepare` calls. A properly written OpenGL program should always allocate free display list and texture names using calls like `glGenLists` and `glGenTextures` (never hardcode display list and texture numbers !), so this shouldn't be ever a problem.

2. Call `RenderBegin` to start actual rendering. This will set up some OpenGL state that will be assumed by further rendering calls. Also, this will do a *push* on OpenGL attributes stack, so that everything can be restored later by `RenderEnd`.

3. Then you should call `RenderShapeState` for each pair of VRML shape node and a state that you want to render. As mentioned earlier, you can only use here state values that you prepared earlier by a `Prepare` call.

   `RenderShapeState` call can be placed inside OpenGL display list, and usually it *should be* to get proper rendering speed.

   Alternatively you can split one `RenderShapeState` call to three calls:
   a. `RenderShapeStateBegin`
   b. `RenderShapeStateNoTransform`
   c. `RenderShapeStateEnd`

   Such splitting is often useful because it allows you to place `RenderShapeStateNoTransform` on a separate display list, that can shared by more shape+state pairs (because it doesn't take some transformations into account).

4. Finally after you rendered all your shapes, you should call `RenderEnd`. As mentioned before, whole OpenGL state will restored by popping OpenGL attribute stack.

   Between `RenderBegin` and `RenderEnd` you are not allowed to change OpenGL state in any way except for calling other `TVRMLOpenGLRenderer` methods. Well, actually there are some exceptions, things that you can legitimately do — these include e.g. setting enabled state of OpenGL blending. But generally you should limit yourself to calling other `TVRMLOpenGLRenderer` methods between `RenderBegin` and `RenderEnd`.

   `RenderBegin` and `RenderEnd` can be wrapped inside OpenGL display lists.

Of course the scenario above may be repeated as many times as you want. The key is that you will not have to repeat `Prepare` calls each time — once a state is prepared, you can use it in `RenderShapeState` calls as many times as you want. If you will not need some state anymore then you can release some resources allocated by it's `Prepare` call by using `UnPrepare` or `UnPrepareAll` methods.

Note that `TVRMLOpenGLRenderer` doesn't try to control whole OpenGL state. It controls only the state that it needs to, to accurately render VRML nodes. Some OpenGL settings that are not controlled include:

- global ambient light value (`glLightModel` with `GL_LIGHT_MODEL_AMBIENT` parameter),
- polygon mode (filled or wireframe ?),
- whether lighting calculation is enabled (although for shapes that VRML specification requires to be unlit `TVRMLOpenGLRenderer` will always temporarily turn lighting calculation off),
- blending settings.

So you can adjust some rendering properties simply by using normal OpenGL commands. Also you can transform rendered VRML models simply by setting appropriate modelview matrix before calling `RenderBegin`. So rendering done by `TVRMLOpenGLRenderer` tries to cooperate with OpenGL nicely, acting just like a "complex OpenGL operation", that plays nicely when mixed with other OpenGL operations.

However, for various implementation reasons, many other VRML rendering properties cannot be controlled by just setting OpenGL state before using `RenderBegin`. Instead you can adjust them by setting `Attributes` property of `TVRMLOpenGLRenderer`.

5.2.1. OpenGL resource cache

Often when you render various VRML models, you will use various `TVRMLOpenGLRenderer` instances. But still you want those `TVRMLOpenGLRenderer` instances to share some common resources. For example, each texture has to be loaded into OpenGL context only once. It would be ridiculous to load the same texture as many times as there are VRML models using it. That's why we have `TVRMLOpenGLRendererContextCache`. It can be used by various renderers to store common resources, like an OpenGL texture name associated with given texture filename.

Things that are cached include:

- Fonts display lists.
- Texture names. This way you can make your whole OpenGL context to share common "texture pool" — and all you have to do is to pass the same `TVRMLOpenGLRendererContextCache` instance around.
- Objects working on higher level than `TVRMLOpenGLRenderer` may also store in the cache various things. Display lists generated for `TVRMLOpenGLRenderer.RenderShapeState` calls are stored in the cache, which allows them to be shared if you started multiple animations from the same scene. Also display lists generated for `TVRMLOpenGLRenderer.RenderShapeStateNoTransform` calls are stored in the cache, this allows them to be shared when multiple animation frames represent the same object but transformed differently.

By default, each `TVRMLOpenGLRenderer` creates and uses his own cache, but you can create `TVRMLOpenGLRendererContextCache` instance explicitly and just pass it down to every OpenGL renderer that you will create. All higher-level objects that use `TVRMLOpenGLRenderer` renderer allow you to pass your desired `TVRMLOpenGLRendererContextCache`. And you should use it, if you want to seriously conserve memory usage of your program.

Also note that when animating, all animation frames of given animation object (`TVRMLGLAnimation` instance, that will be described in details in Chapter 6, *Animation*) always use the same renderer. So they also always use the same cache instance, which already gives you some memory savings thanks to cache automatically.

5.2.2. Specialized OpenGL rendering routines vs Triangulate approach

There are two approaches to render specific VRML nodes within `TVRMLOpenGLRenderer`:

1. The *specialized rendering routines for each node* approach: for all VRML nodes we have routines to render them efficiently using OpenGL. This means that we have to write one routine for each `TNodeGeneralShape` descendant class — one routine to render a `Sphere`, one to render a `Box`, one to render `IndexedFaceSet` and so on. This approach is used by default.

2. The *triangulate approach*. This is an alternative rendering method that will be used if you define `USE_VRML_NODES_TRIANGULATION` symbol for compilation of `VRMLOpenGLRenderer` unit. Each node will be triangulated using `TNodeGeneralShape.LocalTriangulate` method (mentioned earlier in Section 2.7.2, “Triangulating”) and each triangle will be passed to OpenGL.

   This is a proof-of-concept implementation that shows that using `TNodeGeneralShape.LocalTriangulate` we can render all nodes in the same manner — no need to write separate rendering routines for various `TNodeGeneralShape` descendants. All you have to do is to implement triangulating.

   This approach is useful for testing purposes, to test that `LocalTriangulate` methods work correctly. And this allows you to render nodes that don't have specialized rendering procedure done yet. It has a couple of practical disadvantages:

   a. It's slower than dedicated rendering procedures for each node. See discussion below.

   b. Things that are not expressed as triangles (`IndexedLineSet`, `PointSet`) will not be rendered at all.

   c. It lacks some features, because the triangulating routines do not return enough information. For example, textures are not applied (because texture coordinates are not generated), flat shading is always used (because the whole triangle has always only one normal vector). This disadvantage could be removed in the future (by extending information that triangulate callback returns).

By default we use the specialized OpenGL routines for each VRML shape node. Do we want to ever switch to the “triangulate approach” as the default approach ? Clearly the “triangulate approach” has some advantages:

- Extending the triangulate callbacks would give ray-tracer more information. This is the key to implementing normal vectors interpolating and texture mapping in ray-tracer, so it will have to be eventually done anyway.

- OpenGL rendering code would be much shorter. You can already see this by comparing a couple lines of code used by `VRMLOpenGLRenderer` when `USE_VRML_NODES_TRIANGULATION` is defined versus all the files and lines of code used otherwise. In the “triangulate approach” it doesn't matter what `TNodeGeneralShape` descendant you have. All you have to do is to call it's triangulating method and pass all triangles to OpenGL.

But the key problem is that the “triangulate approach” makes OpenGL rendering much less efficient. When writing specialized OpenGL rendering routines we can use OpenGL primitives like `GL_QUAD_STRIP` and such that allow every sensible OpenGL implementation to minimize vertex calculations (in other words, more vertex sharing). In case of indexed shapes (`IndexedFaceSet`) we can even put all vertexes inside vertex array and then lock it (using `GL_EXT_compiled_vertex_array` OpenGL extension, honored by almost every implementation), again greatly improving vertex sharing.

On the other hand current simple triangulator just returns every triangle as a three points, each of them with 3 float values. So it has no way to achieve vertex sharing. To get vertex sharing with triangulator we would have to change our triangulator callbacks to return data in much more complicated way: indexes to arrays instead of direct positions, and a way to produce `GL_QUAD_STRIP` and other OpenGL primitives. But extending triangulator in this direction means that I would have to "expose" all these OpenGL mechanisms for triangulator, complicating greatly not only triangulator interface but also handling of triangulator data by code that doesn't need vertex sharing (like ray-tracer or octree for collision detection). Alternatively, I could make a simpler triangulator (for ray-tracer and such) working on the base of the more general triangulator, but this is also quite some work to implement. This all means some implementation work — as it is now, this would actually make *more complicated implementation than the current "specialized routines for each node" approach*. This may sound like a blasphemy, but in this particular case code duplication (since the simple triangulator routines and specialized OpenGL renderers somewhat duplicate each other algorithms) seems more manageable than alternative of complicated triangulator implementation. At least for now.

Note that even ray-tracer efficiency (and precision) suffers right now because ray-tracer uses the "triangulate approach". That's because ray-tracer could use specialized collision checking for e.g. spheres. Ray collision check with one sphere is much faster (and at the same time more precise) than triangulating sphere and checking for collision with each generated triangle.

5.3. Flat scene for OpenGL

`TVRMLFlatSceneGL` is a descendant of `TVRMLFlatScene` (which was introduced earlier in Section 2.10, "Flat scene"). Internally it uses `TVRMLOpenGLRenderer` (introduced in last section, Section 5.2, "Basic OpenGL rendering") to render scene to OpenGL. It also provides higher-level optimizations and features for OpenGL rendering. In short, this is the most comfortable and complete class that you should use to load and render static VRML models. In addition to `TVRMLFlatScene` features, it allows you to:

- Render all shape+state pairs (i.e. whole VRML scene). Use `Render` method with `nil` as `TestShapeStateVisibility` parameter for the simplest rendering method.

- You can render only the shape+state pairs that are within current camera frustum by `RenderFrustum`. This works by checking each shape+state pair for collision with frustum before rendering. Generally, it makes a great rendering optimization if user doesn't usually see the whole scene at once.

- An even better choice than `RenderFrustum` is `RenderFrustumOctree`. This works like `RenderFrustum`, but the shape+state pairs within the frustum are determined by traversing the shape+state octree. If your scene has many shape+state pairs then this will be faster than normal `RenderFrustum`.

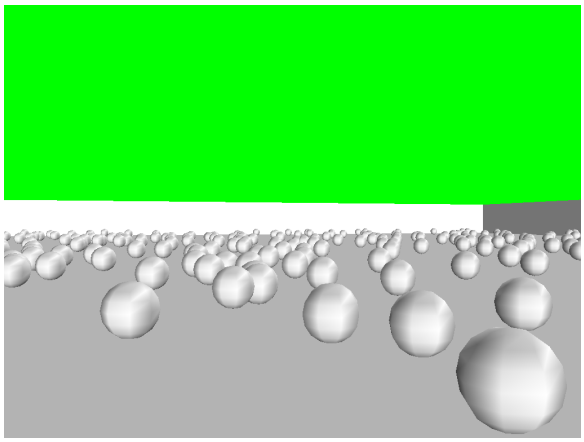
- In special cases you may be able to create a specialized test whether given shape+state pair is visible. You can call `Render` method passing as a parameter pointer to your specialized test routine. This way you may be able to add some special optimizations in particular cases.

  For example if you know that the scene uses a dense fog and it has a matching background color (for example by `Background` VRML node) then it's sensible to ignore shape+state pairs that are further then fog's visibility range. In other words, you only draw shapes within a sphere around the player position.

A working example program that uses exactly this approach is available in our engine sources in the file `units/3dmodels.gl/examples/fog_culling.dpr`.

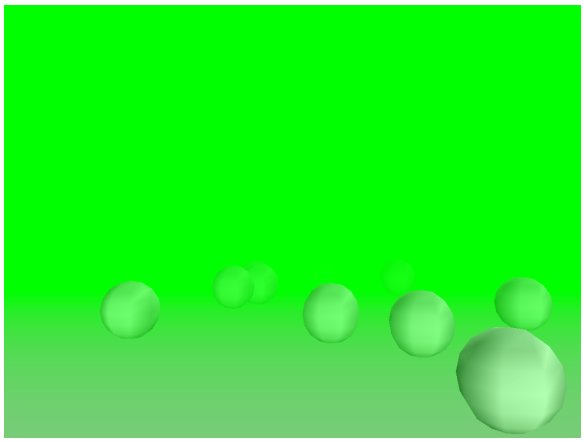
On the screenshot below the fog is turned off. Camera frustum culling is used to optimize rendering, and so only 297 spheres out of all 866 spheres on the scene need to be rendered.

Figure 5.1. Rendering without the fog (camera frustum culling is used)



On the next screenshot the fog is turned on. The same view is rendered. We render only the objects within fog visibility range, and easily achieve a drastic improvement: only 65 spheres are passed to OpenGL now. Actually we could improve this result even more: in this case, both camera frustum culling and culling to the fog range could be used. Screenshot suggests that only 9 spheres would be rendered then.

Figure 5.2. Rendering with the fog (only objects within the fog visibility range need to be rendered)



- `TVRMLFlatSceneGL` implements material transparency by OpenGL alpha blending. This requires rearranging the order in which shape+state pairs are rendered, that's why it must be done in this class (instead of being done inside `TVRMLOpenGLRenderer`).

Details about this will be revealed soon in Section 5.3.1, “Material transparency using OpenGL alpha blending”.

- `TVRMLFlatSceneGL` automatically builds and uses display lists to optimize rendering of the VRML model. You can choose the “granularity” of display list creation by the `Optimization` property. This is a very important optimization that in practice greatly speeds up OpenGL rendering.

  Details about this will be revealed soon in Section 5.3.3, “Display lists strategies”.

- `TVRMLFlatSceneGL` has also comfortable methods to handle and render VRML `Background` node of your scene.

Note that `TVRMLFlatSceneGL` caches various information, just like `TVRMLFlatScene`, as discussed in Section 2.10.3, “Caching”. Cached things include `Background` node information and created display lists. In case of display lists it's sort-of obvious that they are cached, since the very idea of display lists is to create them once and reuse many times already stored calculation. In practice, caching means that you must call some `ChangedXxx` method if you change something in your VRML node graph after loading it.

5.3.1. Material transparency using OpenGL alpha blending

To understand the issue you have to understand how OpenGL works. OpenGL doesn't “remember” all the triangles sent to it. As soon as you finish passing a triangle to OpenGL (which means making `glVertex` call that completes the triangle) OpenGL implementation is free to immediately render it. This means mapping the given triangle to 2D window and updating data in various buffers — most notably the color buffer, but also the depth buffer, the stencil buffer and possibly others. Right after triangle is rendered this way, OpenGL implementation can completely “forget” about the fact that it just rendered the triangle. All triangle geometry, materials etc. information doesn't have to be kept anywhere. The only trace after rendering the triangle is left in the buffers (but these are large 2D arrays of data, and only the human eye can reconstruct the shape of the triangle by looking at the color buffer contents).

In summary, this means that the order in which you pass the triangles to OpenGL is significant. Rendering opaque objects with the help of depth buffer is the particular and simple case when this order doesn't matter (aside for issues related to depth buffer inaccuracy or overlapping geometry). But generally the order matters. Using alpha blending is one such case.

To implement VRML material transparency we use materials with *alpha* (4th color component) set to value lower than 1.0. When the triangle is specified, OpenGL renders it. A special operation mode is done for updating color buffer: instead of overriding old color values, the new and old colors are mixed, taking into account alpha (which acts as opacity factor here) value. Of course when rendering transparent triangles they still must be tested versus depth buffer, that contains at this point information about *all the triangles rendered so far within this frame*.

Now observe that depth buffer should not be updated as a result of rendering partially transparent triangle. Reason: partially transparent triangle doesn't hide the geometry behind it. If we will happen to render later other triangle (partially transparent or opaque) behind current partially transparent triangle, then the future triangle should not be eliminated by

the current triangle. So only rendering opaque objects can change depth buffer data, and thus opaque objects hide all (partially transparent or opaque) objects behind them.

But what will happen now if you render opaque triangle that is behind already rendered partially transparent triangle ? The opaque triangle will cover the partially transparent one, because the information about partially transparent triangle was not recorded in depth buffer. For example you will get this incorrect result:

Figure 5.3. The ghost creature on this screenshot is actually very close to the player. But it's transparent and is rendered incorrectly: gets covered by the ground and trees.

The solution is to avoid this situation and *render all partially transparent objects after all opaque objects*. This will give correct result, like this:

Figure 5.4. The transparent ghost rendered correctly: you can see that it's floating right before the player.

Actually, in a general situation, rendering all partially transparent objects after opaque objects is not enough. That's because if more than one transparent object is visible on the same screen pixel, then the order in which they are rendered matters — because they are blended with color buffer in the same order as they are passed to OpenGL. For example if you set your blending functions to standard (`GL_SRC_ALPHA`,

GL_ONE_MINUS_SRC_ALPHA) then each time you render a triangle with color (Red, Green, Blue) and opacity α, the current screen pixel color ($Screen_{Red}$, $Screen_{Green}$, $Screen_{Blue}$) changes to

$$(Screen_{Red}, Screen_{Green}, Screen_{Blue}) * (1 - \alpha) + (Red, Green, Blue) * \alpha$$

Consider for example two partially transparent triangles, one of them red and the second one green, both with α set to 0.9. Suppose that they are both visible on the same pixel. If you render the red triangle first, then the pixel color will be

$$ScreenColor * (1 - \alpha) * (1 - \alpha) + RedColor * \alpha * (1 - \alpha) + GreenColor * \alpha =$$
$$ScreenColor * 0.01 + RedColor * 0.09 + GreenColor * 0.9 =$$
visible as GreenColor in practice

If you render green triangle first then the analogous calculations will get you pixel color close to the red.

So the more correct solution to this problem is to sort your transparent triangles with respect to their distance from the viewer. You should render first the objects that are more distant.

However, this solution isn't really nice. Sorting all triangles at each frame (or after each camera move) doesn't seem like a good idea for a 3D simulation that must be done in real-time as fast as possible. Moreover, there are pathological cases when even sorting is not enough and you will have to split triangles to get things 100% right. Of course, you could sort larger objects (for example whole VRML shape nodes) instead of triangles to make the process faster. But then you may have to split VRML shape nodes sometimes (in addition to splitting triangles inside in pathological cases), or agree to non-perfect results.

That's why our engine (as well as many other OpenGL rendering engines) just ignores the sorting problem. We do not pay any attention to the order of rendering of transparent objects — as long as they are rendered after all opaque objects. In practice, rendering artifacts will occur only in some complex combinations of transparent objects. If you seldom use a transparent object, then you have small chance of ever hitting the situation that actually requires you to sort the triangles. Moreover, even in these situations, the rendering artifacts are usually not noticeable to casual user. Fast real-time rendering is far more important that 100% accuracy here.

Moreover, our engine right now by default uses (GL_SRC_ALPHA, GL_ONE) blending functions, which means that the resulting pixel color is calculated as

$$(Screen_{Red}, Screen_{Green}, Screen_{Blue}) + (Red, Green, Blue) * \alpha$$

That is, the current screen color is not scaled by $(1 - \alpha)$. We only add new color, scaled by it's alpha. This way rendering order of the transparent triangles doesn't matter — any order will produce the same results. For some uses (GL_SRC_ALPHA, GL_ONE) functions look better than (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA), for some uses they are worse. (GL_SRC_ALPHA, GL_ONE) tend to make image too bright (since transparent objects only increase the color values), that's actually good as long as your transparent objects represent some bright-colored and dense objects (a thick plastic glass, for example). (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) on the other hand can sometimes unnaturally darken the opaque objects behind (since that's what these functions will do for a dark transparent object with large alpha).

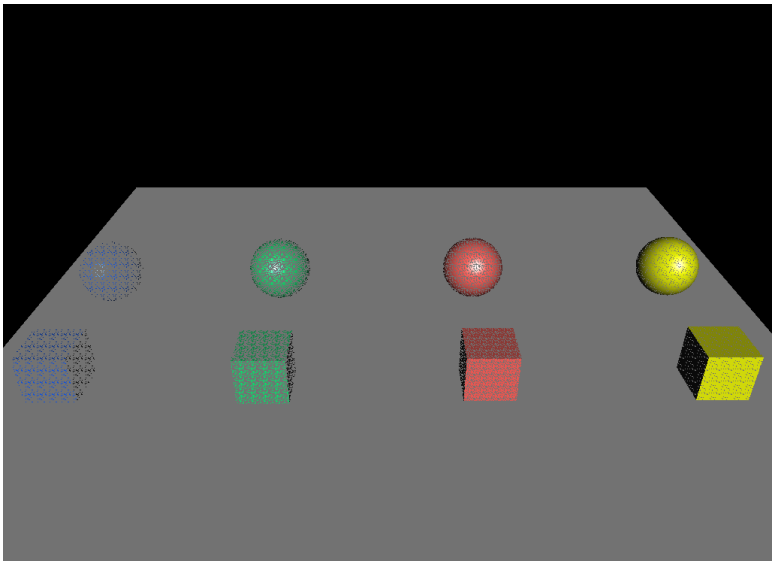
5.3.2. Material transparency using polygon stipple

Other method of rendering material transparency deserves a quick note here. It's done by

polygon stipple, which means that transparent triangles are rendered using special bit mask. This way part of their pixels are rendered as opaque, and part of them are not rendered at all. This creates a transparent look on sufficiently large resolution. Order of rendering transparent objects doesn't matter in this case.

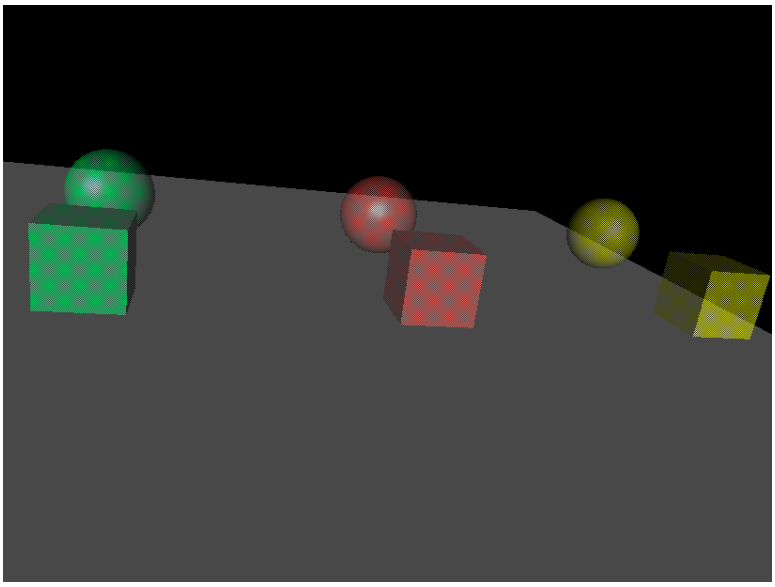
However, the practical disadvantages of this method is that it looks quite, well, ugly. When we use random stipples (to precisely show different transparency of different objects) then the random stipples look very ugly:

Figure 5.5. Material transparency with random stipples



Instead of using random stipples, we can use a couple of special good-looking prepared regular stipples. But then we don't have much ability to accurately represent various transparency values (especially for very transparent objects). And still the results look quite bad:

Figure 5.6. Material transparency with regular stipples



5.3.3. Display lists strategies

`TVRMLFlatSceneGL` allows you to choose one of the four display lists optimization methods. Which method is the best really depends on various factors, including how often you will modify your VRML nodes graph while rendering and how user will usually view the scene. If in doubt, a good default is to choose the `roSeparateShapeStates` method.

view3dscene [http://www.camelot.homedns.org/~michalis/view3dscene.php] allows you to test all optimization methods by `--renderer-optimization` command-line option.

The available optimization methods are:

1. **`roNone`.** This stands for “no optimization”. No OpenGL display lists will be ever constructed. This means that rendering is always done by real OpenGL calls (through rendering methods of `TVRMLOpenGLRenderer`). So rendering may be slow.

   On the other hand, changing your VRML nodes graph at run-time doesn't hurt. Calls to methods like `PrepareRender` and `ChangedAll` are very fast (as they will act almost as null operations).

   Use this optimization method if you plan to change the scene at run-time very often (for example, at each `OnIdle` event). Building display lists in such case would be only a waste of time, since they would have to be rebuild very often. Alternatively, this is useful optimization if you have a special situation are you know that you will render given `TVRMLFlatSceneGL` scene very few times (for example, only 1 or 2 times).

2. **`roSceneAsAWhole`.** Treat the scene as a one big static object. One OpenGL display list will be created, that renders the whole object. This is a great optimization method if the scene is static (otherwise rebuilding display lists too often would cost too much time) and you're sure that user will usually see the whole scene (or at least a large part

of it).

For example, this is suitable optimization for an `Examine` navigation method, when the whole scene is displayed. Or for small VRML models, like creatures or items in a 3D game — such models are usually either visible completely or not visible at all. So there is no need for any more sophisticated optimization. Although if the creatures are animated, you should choose other optimization method (preferably `roSeparateShapeStatesNoTransform`) to conserve memory usage.

If the scene is static but user usually only looks at some small part of it (for example, a typical 3D game level; or a `Walk` navigation method) then this optimization is not a good choice. That's because this optimization nullifies the purpose of `RenderFrustum` and `RenderFrustumOctree` methods. And the purpose of the `Render` method with a non-nil function to test shape visibility. In case of this optimization all render methods will always render the whole scene to OpenGL.

3. **`roSeparateShapeStates`.** Build separate OpenGL display list for each shape+state pair. This is a good optimization method if:

   a. You're not going to modify anything within the VRML graph after loading the scene, or you will modify only some local parts of it (that affect only small part of shape+state pairs). In the latter case, we will rebuild, on change, only display lists for the affected shape+state pairs.

   b. You know that usually user will not see the whole scene, only a small part of it. In this case you want to render the scene using methods like `RenderFrustum` or `RenderFrustumOctree`, and this optimization allows them to work perfectly: only the needed shape+state pairs will be passed to OpenGL.

   c. Another advantage of this optimization comes when you use `TVRMLGLAnimation` (discussed in details in later Chapter 6, *Animation*). If a large part of your animation is actually still (i.e. the same shape+state is used, with the same nodes inside), and only the other part animates, then the still shape+states pairs will actually use and share only one display list. Only one — throughout all the time when they are still! This can be a huge memory saving, which is important because generating many display lists for `TVRMLGLAnimation` is generally very memory-hungry operation.

      Actually you may be able to achieve even better memory savings by using `roSeparateShapeStatesNoTransform` optimization method discussed next.

      Such display list sharing is possible thanks to `TVRMLOpenGLRendererContextCache` that caches display lists and allows them to be reused by various animation frames.

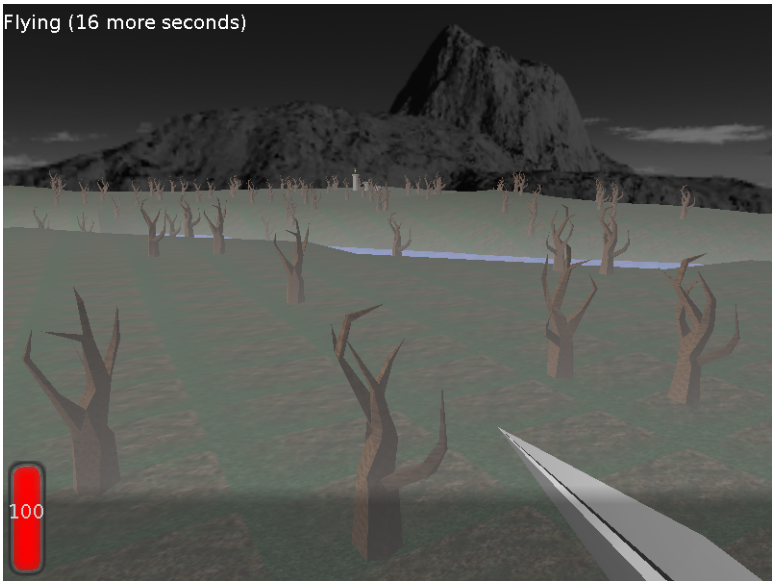
4. **`roSeparateShapeStatesNoTransform`.** This is like `roSeparateShapeStates` but it allows for much more display lists sharing because it stores untransformed shape+state in a display list. In other words, it wraps `RenderShapeStateNoTransform` call of `TVRMLOpenGLRenderer` instead of the whole `RenderShapeState`.

   When this is better over `roSeparateShapeStates`:

   a. If you use `TVRMLGLAnimation` when the same shape+state occurs in each animation frame but differently transformed. For example, a robot moves by bending it's legs at the knees. But the thighs and the calves' shapes remain the same, only the transformations of the calves change.

b. When you have a scene that uses the same shape+state many times but with different transformation. For example a forest using the same tree models scattered around. In this case only one display list to render the tree is created, and this can be a huge memory saving if we have many trees in our forest.

Figure 5.7. All the trees visible on this screenshot are actually the same tree model, only moved and rotated differently.



Actually, “transformation” here means everything rendered by `TVRMLOpenGLRenderer.RenderShapeStateBegin` method. This includes not only the modelview transformation, but also the texture transformation and all lights settings. So there are even more cases when `roSeparateShapeStatesNoTransform` will use one display list, while `roSeparateShapeStates` would use a lot. Sometimes this can be a *huge memory saving*. Also preparing scene/animations (preparing their display lists, e.g. by `PrepareRender` call or implicitly during the first `Render` call) should be much faster.

Unfortunately, `roSeparateShapeStatesNoTransform` has some disadvantages when compared with `roSeparateShapeStates`:

- `roSeparateShapeStatesNoTransform` can be used only if you don't use `Attributes.OnBeforeVertex` feature and your model doesn't use volumetric fog. If you do use these features, you have no choice: you must use `roSeparateShapeStates`, otherwise rendering results may be wrong.

  For example, look at these two trees on a scene that uses the blue volumetric fog.

Figure 5.8. The correct rendering of the trees with volumetric fog. Using `roSeparateShapeStates` optimization.

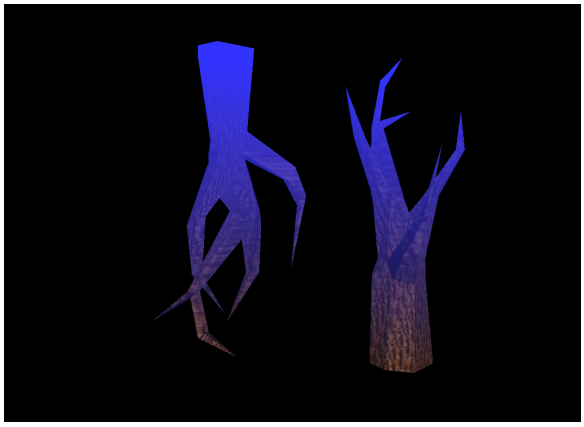
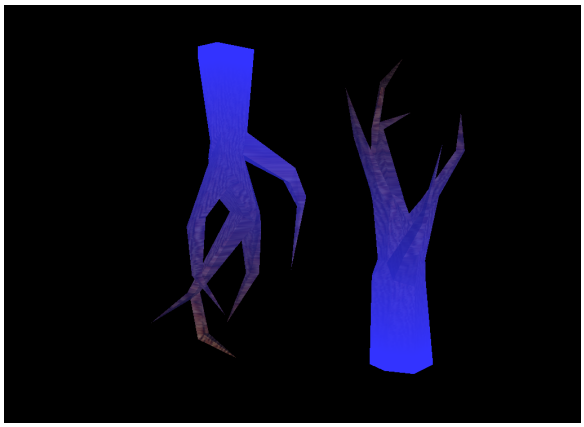


Figure 5.9. The wrong rendering of the trees with volumetric fog. Using `roSeparateShapeStatesNoTransform` optimization.



The reason of this problem: `roSeparateShapeStatesNoTransform` assumes that the work done by `RenderShapeStateNoTransform` call of `TVRMLOpenGLRenderer` really doesn't depend on current transformation. However, for OpenGL volumetric fog vertex parameter, and for calculating `Attributes.OnBeforeVertex` callback parameter, we have to use current transformation.

- In some cases `roSeparateShapeStatesNoTransform` may be a little slower at rendering than `roSeparateShapeStates`, as it doesn't wrap in display list things done by `TVRMLOpenGLRenderer.RenderShapeStateBegin` method. So modelview matrix and texture matrix and whole lights state will be applied each time by “real” OpenGL commands, without display lists.

  Will this affect rendering speed much ? If your scene doesn't use lights then the speed difference between `roSeparateShapeStates` and `roSeparateShapeStatesNoTransform` should not be noticeable. Otherwise... well, you

have to check what matters more in this case: memory saving by `roSeparateShapeStatesNoTransform` or additional speed saving by `roSeparateShapeStates`.

5.3.3.1. Various optimizations for the same VRML model

Note that you are free to create more than one `TVRMLFlatSceneGL` instance for the same VRML nodes graph. Just be careful to:

- Do not try to free the same `RootNode` object more than once. Setting `OwnsRootNode` property of `TVRMLFlatSceneGL` to false allows you to take manual control over when your VRML nodes graph will be freed.
- Also you should pass the same `TVRMLOpenGLRendererContextCache` instance to various `TVRMLFlatSceneGL` instances, to share OpenGL resources.
- In fact, you can even force various `TVRMLFlatSceneGL` instances to use the same `TVRMLOpenGLRenderer`, but this is more tricky.

Using this approach you can even create various `TVRMLFlatSceneGL` instances that render the same VRML model but with different optimizations.

5.3.4. Shape+state granularity

Optimizations done by `TVRMLFlatSceneGL` (in particular, frustum culling) work best when the scene is sensibly divided into a number of small shape+state pairs. This means that “internal” design of VRML model (how it's divided into shape+states) matters a lot. Here are some guidelines for VRML authors:

- Don't define your entire world model as one `IndexedFaceSet` node (since this will effectively reduce `roSeparateShapeStatesNoTransform` and `roSeparateShapeStates` optimization methods to being equivalent to `roSceneAsAWhole`).
- Avoid `IndexedFaceSet` nodes with triangles that are scattered all around the whole scene. Such nodes will have very large bounding box and will be judged as visible from almost every camera position in the scene, thus making optimizations like frustum culling less efficient.
- An ideal VRML model is split into many shape nodes that have small bounding boxes. It's hard to specify a precise “optimal” number of shape nodes, so you should just test your VRML model as much as you can. Generally, `RenderFrustumOctree` should be able to handle efficiently even models with a lot of shape+state pairs.

5.3.4.1. Triangle granularity ?

Then comes an idea to use scene division into triangles instead of shape+state pairs. This would mean that our optimization doesn't depend on shape+state division so much. Large shape+states would no longer be a problematic case.

To make this work we would have to traverse triangle octree to decide which triangles are

in the visibility frustum. Doing this without the octree, i.e. testing each triangle against the frustum, would be pointless, since this is what OpenGL already does by itself.

Such traversing of the octree would have to be the first pass, used only to mark visible triangles. In the second pass we would take each shape+state pair and render marked triangles from it. The reason for this two-pass approach is that otherwise (if we would try to render triangles immediately when traversing the octree) we would produce too much overhead for OpenGL. Overhead would come from changing material/texture/etc. properties very often, since we would probably find triangles from various nodes (with various properties) very close in some octree leafs.

But this approach creates problems:

- The rendering routines would have to be written much more intelligently to avoid rendering unmarked triangles. This is not as easy as it seems as it collides with some smart tricks to improve vertex sharing, like using OpenGL primitives (`GL_QUAD_STRIP` etc.).
- We would be unable to put large parts of rendering pipeline into OpenGL display lists. Constructing separate display list for each triangle has little sense.

That's why this approach is not implemented.

Chapter 6. Animation

`TVRMLGLAnimation` class is used to build and render animations. When constructing it's instance you provide one or more VRML models that have exactly the same structure, but may have different values for various fields. For each provided model you specify an associated position in time. The resulting animation will change the first VRML model to the last one, such that at any time point we will either use one of the predefined models (if point in time is close to the model's associated time) or a new model created by interpolating between two successive models in time.

For example, the first model may be a small sphere with blue color, and the second model may be a larger sphere with white color. The resulting animation depicts a growing sphere with color fading from blue to white.

Every animation may be played backwards and/or in a loop.

The models must be the same structurally, but most of the field types may have different values. VRML field types that are allowed to differ include `SFColor`, `SFFloat`, `SFMatrix`, `SFRotation`, `SFVec2f` and `SFVec3f`. Equivalent multi-valued fields may have different values too (but still must have the same number of items). This gives you an endless list of possibilities what can be expressed as an animation. Some examples:

- Moving, rotating, scaling objects may be expressed by changing transformation values.
- Any kind of morphing (mesh deformation) may be expressed by changing values of `IndexedFaceSet` coordinates.
- Materials, colors, lights may change. Even such properties like a material transparency, or a light position or direction.
- Texture coordinates may change to achieve effects like a moving water surface.

If you want to experiment with animations the sources of our engine contain a program `demo_animation` (see the file `units/3dmodels.gl/examples/demo_animation.dpr`), and various sample models for animating (see subdirectory `models/`). Also "The Castle" [http://www.camelot.homedns.org/~michalis/castle.php] uses animations for all creatures and weapons.

6.1. How does it work

First of all, for now the scenes are *not interpolated when rendering*. Instead, at construction time, we create a number of new interpolated models and save them (along with the models that were specified explicitly). The property `ScenesPerTime` says with what granularity the intermediate scenes are constructed for a time unit.

If you specify too large `ScenesPerTime` your animations will take a lot of time to prepare and will require a lot of memory. On the other hand too small `ScenesPerTime` value will result in an unpleasant jagged animation. Ideally, `ScenesPerTime` should be >= than the number of frames you will render in your time unit, but this is usually way too large value.

Internally, the `TVRMLGLAnimation` wraps each model (that was specified explicitly or created by interpolation) in a new `TVRMLFlatSceneGL` instance. This means that we

have all the features of our static OpenGL rendering available when doing animations too. The suggested display list optimization for animations is usually `roSeparateShapeStatesNoTransform`, since this allows various animation frames to share as much display lists as they can. Sharing display lists is very important for animations, otherwise you can easily run out of memory (and preparing animations will take a long time).

6.2. Comparison with VRML interpolator nodes

VRML 97 has special interpolator nodes [http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/part1/concepts.html#4.6.8] to express animation in a VRML file. Their idea of work is similar to ours, that is animation is done by interpolating some fields values. In general, VRML approach is obviously cleaner: it's better to keep information about interpolating in one file instead of creating two copies only to express an animation.

Moreover, our current approach has a serious deficiency if your animation tries to change two pieces of your model with drastically different speeds. Consider this:

1. It's trivially easy to create an animation with a box that blinks (changes color) 100 times per time unit.
2. It's also trivially easy to create an animation with a sphere that blinks only once for a given time unit.
3. But if you want to create an animation that contains both the box (blinking 100 times/time unit) and the sphere (blinking once for a time unit), you will have to prepare 100 VRML files to express this !

VRML interpolators don't have this problem.

Still we may point some small advantages of our approach over VRML interpolators :

- VRML interpolators don't allow to animate textures coordinates (expressed as `MFVec2f` field type).
- The practical advantage of our approach is that you can design your animations using any authoring software that can export static VRML files. If your modeller can design animations, but doesn't save them to VRML interpolator nodes, all you have to do is to export your models a couple of times from a couple of different points in time.

  Author of this document is not familiar with any professional open-source 3D modelling software that can export animations to VRML interpolator nodes. In particular, VRML 97 exporter from my favourite Blender [http://www.blender3d.org/] cannot do it.

6.3. Future plans

6.3.1. Perform interpolation at rendering time

This will be much less memory-demanding, and will remove the problems with jagged animations because of too small `ScenesPerTime` values.

But note that this will necessarily be either slower or more limited than our current approach:

1. If we want to keep all our features, we will have to use `roNone` optimization (no display list). And all traversing and calculations needed by interpolation will have to be done at rendering time.

2. The other approach is to extend possibilities of `roSeparateShapeStatesNoTransform` optimization. This allows me to easily interpolate things like modelview transformation while keeping everything important inside display lists. In fact, this was the initial idea behind implementing `roSeparateShapeStatesNoTransform` optimization. Right now, `roSeparateShapeStatesNoTransform` optimization allows for display list sharing (which means that memory saving is already achieved). By actually interpolating transformation at rendering time we will achieve the second advantage: no jagged animations.

   However, this will allow us to animate only a limited set of properties. For example animating a whole mesh will not be possible with this approach.

6.3.2. Handling of VRML interpolator nodes

All rendering features of VRML interpolators are available in our engine. To handle interpolator nodes we have to read their key time points and then build appropriate interpolated nodes.

Chapter 7. Links

7.1. VRML specifications

- VRML 1.0 specification [http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html]
- VRML 2.0 (also called VRML 97) specifications [http://www.web3d.org/x3d/specifications/vrml/]
- The Annotated VRML 97 Reference [http:// accad. osu. edu/ ~pgerstma/ class/ vnv/ resources/info/AnnotatedVrmlRef/Book.html]
- X3D specifications [http://www.web3d.org/x3d/specifications/]

7.2. Author's resources

My homepage [http://www.camelot.homedns.org/~michalis/], including:

- VRML engine documentation [http:// www. camelot. homedns. org/ ~michalis/ vrml_engine_doc.php] — the document that you're reading right now
- view3dscene [http://www.camelot.homedns.org/~michalis/view3dscene.php] — VRML 1.0, 2.0, 3DS, OBJ browser
- rayhunter [http://www.camelot.homedns.org/~michalis/rayhunter.php] — command-line ray-tracer, and it's gallery [http:// www. camelot. homedns. org/ ~michalis/ raytr_gallery. php]
- sources of my engine [http:// www. camelot. homedns. org/ ~michalis/ sources. php] and their documentation [http://www.camelot.homedns.org/~michalis/sources_docs.php]
- VRML implementation status [http:// www. camelot. homedns. org/ ~michalis/ vrml_implementation_status.php]
- VRML test suite [http:// www. camelot. homedns. org/ ~michalis/ kambi_vrml_test_suite. php]
- Specification of my extensions to VRML [http://www.camelot.homedns.org/~michalis/ kambi_vrml_extensions.php]