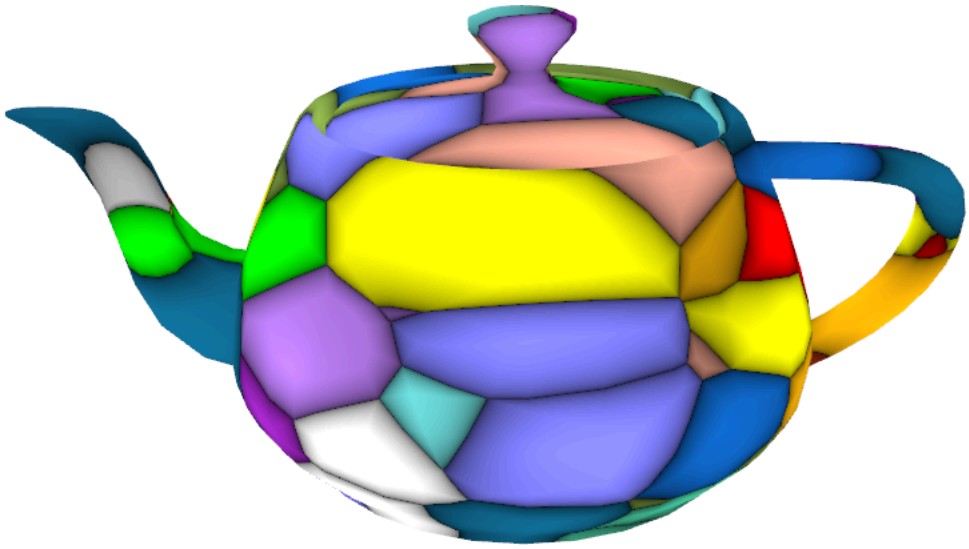
Compositing Shaders in X3D

Michalis Kamburelis



Ph.D. Thesis

Institute of Computer Science
University of Wrocław

Wrocław, 2011-2022

Compositing Shaders in X3D

Michalis Kamburelis

This documentation is open-source and we welcome pull requests to improve it. [https://github.com/castle-engine/cge-documentation/]

Table of Contents

List of Figures

Abstract

We present a new approach for implementing effects using the GPU shading languages. Our effects seamlessly cooperate with each other and with the shaders used internally by the 3D application. Thus the effects are reusable, work in various combinations and under all lighting and texture conditions. This makes the GPU shaders more useful for 3D content authors.

Our approach may also be used to integrate internal effects inside a 3D renderer. Modern renderers need to combine many effects, like lighting, bump mapping and shadow maps. As such, it becomes important to develop all these internal effects easily and separately.

We have designed our effects to fit naturally in 3D scene graph formats, in particular we present a number of extensions to the X3D standard. Our extensions nicely integrate shader effects with X3D concepts like shapes, groups, light sources and textures.

Chapter 1. Overview

X3D [X3D] is an open standard for representing interactive 3D models, with many advanced graphic features. Chapter 2, *What is X3D?* describes X3D in more detail.

Programmable shaders component [X3D Shaders] (part of the X3D standard) defines how *shaders* can be assigned to visible shapes. *Shaders* are programs usually executed on the graphic processor unit (GPU). They control the per-vertex and per-pixel processing, for example summing the lights contribution and mixing the texture colors. The authors can create and assign shaders to shapes, which makes a myriad of interesting graphic effects possible in X3D models. Chapter 3, *Shaders and X3D* describes shaders and the standard way to use them with X3D.

The shaders designed using the standard nodes *replace* the normal rendering functionality, not *enhance* it. This reflects the underlying API, like OpenGL or Direct3D. The 3D libraries, in turn, follow the hardware idea that shader code should be a complete and optimized program designed for rendering a particular shape.

We argue that a different approach is needed in many situations. Authors usually would like to keep the normal rendering features working and only add their own effects. The 3D renderer already has an extensive internal shaders system and the authors want to depend on these internal shaders to do the common job.

As an example, consider this simplified lighting equation:

$$\sum_{l \in Lights} shadow(l) * light\_color(l, material, normal(point))$$

Different effects want to change various parts of this equation, without touching the others. For example, the *shadow* function may check a shadow map pixel, or (when shadow map is not available) always return 1. The *normal* function may take the vector straight from the geometry description, or calculate it using a texture value (classic bump mapping). See Figure 1.1, “Japanese shrine model with more and more effects applied”. The *light_color* function may use different lighting models (Phong, Ward, Cook-Torrance and so on). Sometimes it makes sense to change these functions for all the light sources and sometimes only a specific light source should behave differently. Our approach allows you to do everything mentioned here.

We present a system for creating effects by essentially compositing pieces of a shader code. All the effects defined this way effortlessly cooperate and can be combined with each other and with application internal shaders. This makes shader programs:

1. Much easier to create. We can jump straight into the implementation of our imagined algorithm in the shader. We are only interested in modifying the relevant shader calculation parameter. We do not need to care about the rest of the shader.

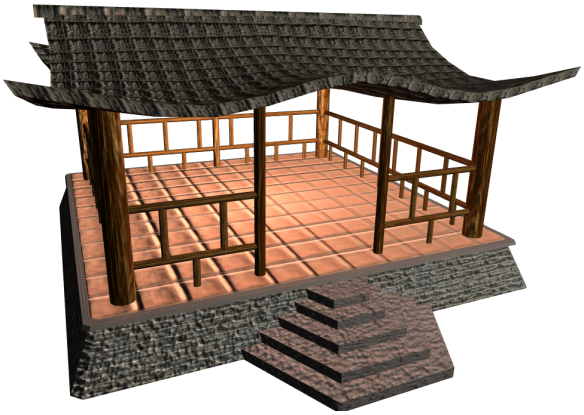
2. Much more powerful. Our effect immediately cooperates with absolutely every normal feature of X3D rendering. This makes the implemented effect useful for a wide range of real uses, not only for a particular situation or a particular model (as it often happens with specialized shader code). All X3D light sources, textures, even other shader effects, are correctly applied.

It is important to note that we keep the full power of a chosen GPU shading language. We deliberately do not try to invent here a new language, or wrap existing language in some cumbersome limitations. This is most flexible for authors, and it also allows an easy implementation – there is no need for any complex shading language processing inside the application.

Figure 1.1. Japanese shrine model with more and more effects applied. The model is based on http://opengameart.org/content/shrine-shinto-japan.

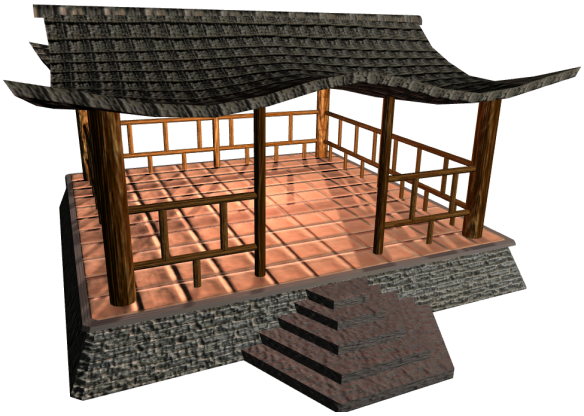
No effects.

Phong shading (per-pixel lighting).



Bump mapping.

1st shadow map.



2nd shadow map.

Both shadow maps.

Chapter 2. What is X3D?

X3D [X3D] is a language to describe 3D worlds. The precise specification of the language is open to everyone. In effect, many applications can handle X3D and cooperate with each other. For example, you can create an X3D file using any popular 3D modeller (like *Blender*) and then load it into any X3D browser (like our *view3dscene*, see [Castle Game Engine]).

Many common 3D features, like triangle meshes with materials and textures, are easy to express. Yet the whole X3D standard is quite large, including advanced 3D concepts like NURBS, cube mapping, multi-texturing, particle effects, skinned humanoid animation, spatial sound, and physics.

The scene is represented as a graph of *nodes*. In the simple cases, the scene is just a tree of nodes. In a general case, it can be a directed graph of nodes, with possible cycles. The X3D specification lists the available node types and their fields. It is also possible to define new full-featured node types using *prototypes*.

An example of a simple X3D file in the "classic" encoding follows. You can save it as a file named `test.x3dv` and open with any X3D browser.

```
#X3D V3.2 utf8
PROFILE Interchange
Shape {
  geometry Sphere { radius 2 }
}
```

Example below shows the same X3D content encoded using the XML format. Note that we omitted DTD and XML schema declarations for brevity. Again, you can open this file with any X3D browser (be sure to save it with an `.x3d` extension).

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D version="3.2" profile="Interchange">
  <Scene>
    <Shape>
      <Sphere radius="2" />
    </Shape>
  </Scene>
</X3D>
```

To enable basic animation and interactive behavior, X3D introduces the concept of *events* and *routes*. Many nodes have the ability to *send events*, notifying about something. There are even special nodes called *sensors* with the sole purpose of sending events when something happens. For example *mouse sensors*, that report user clicking and dragging on the scene. Independently, many nodes can also *receive events*, which allows to instruct the node to do something (for example, start the animation). The X3D author can declare *routes* that connect given node's *output event* (a socket used to send an event) to another node's *input event* (a socket used to receive an event). For example, "open a door when the handle is pressed".

Below is an example of a simple interactive animation. When you click on a sphere, the `TimeSensor` starts ticking, which in turn makes the `PositionInterpolator` produce 3D positions with increasing Y value. The positions are then used to move the sphere up.

```
#X3D V3.2 utf8
PROFILE Interchange
```

```
DEF MyTransform Transform {
  children Shape {
    geometry Sphere { }
  }
}

DEF MyTouchSensor TouchSensor { }

DEF MyTimeSensor TimeSensor { }

DEF MyInterpolator PositionInterpolator {
  key      [ 0      1     ]
  keyValue [ 0 0 0  0 1 0 ]
}

ROUTE MyTouchSensor.touchTime TO MyTimeSensor.startTime
ROUTE MyTimeSensor.fraction_changed TO MyInterpolator.set_fraction
ROUTE MyInterpolator.value_changed TO MyTransform.set_translation
```

Chapter 3. Shaders and X3D

Graphic processing unit (GPU) is given a set of 3D points (vertexes) connected into triangles. For each vertex, some work should be performed, at least to transform it from an object space into the clip space. This is where we move and rotate our objects, and apply perspective projection. *Vertex shaders* allow to replace this per-vertex work with a custom program written in a special *shading language*. When the vertexes are processed, the GPU performs *rasterization*, determining which screen pixels are actually covered by the triangles. Then each pixel is drawn, which involves calculating the actual pixel color. For example we can mix color from the lighting calculations with the texture color. *Fragment (pixel) shaders* allow to replace this per-pixel work with a custom program.

An optional *geometry shader* may also change the primitives between the vertex and fragment processing. We will talk about geometry shaders more in Chapter 6, *Extensions for geometry shaders*.

The most popular real-time shading languages right now are OpenGL *GLSL* [GLSL], NVidia *Cg* [Cg] and Direct 3D *HLSL* [HLSL]. They are used for the same purposes and offer practically the same possibilities. X3D, and our extensions for compositing shaders described in this paper, support all three of these languages.

The current implementation of our extensions supports only the *GLSL* (*OpenGL Shading Language*), which is probably the most natural to use in an engine based on OpenGL. As such, most of our examples in this paper will show *GLSL*.

Example *GLSL* vertex shader and accompanying fragment shader:

```
/* vertex shader */
void main(void)
{
  /* pass unchanged texture coordinate to the fragment shader */
  gl_TexCoord[0] = gl_MultiTexCoord0;
  /* calculate vertex position in clip space */
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
/* fragment shader */
/* myTexture contents are loaded outside of the shader program,
   by appropriate OpenGL calls. */
uniform sampler2D myTexture;
void main(void)
{
  /* take the texture color at given coordinates, multiply by 2,
     use it as the fragment (pixel) color */
  gl_FragColor = texture2D(myTexture, gl_TexCoord[0].st) * 2.0;
}
```

The shader source code should be processed and passed to the rendering 3D library, like OpenGL, that in turn will pass it to the hardware (GPU). The complexity of this operation (and the differences between various shading languages at this step) can be fortunately completely ignored by us. That is because standard X3D *Programmable shaders component* [X3D Shaders] gives us a simple way to attach a shader source code to a 3D shape. The X3D browser will do all the necessary job of handling the shader to the underlying libraries and hardware.

A simple working example showing X3D with *GLSL* shader code:

```
#X3D V3.2 utf8
PROFILE Interchange
Shape {
  appearance Appearance {
    shaders ComposedShader {
      language "GLSL"
      parts ShaderPart {
        type "FRAGMENT"
        url "data:text/plain,
          void main(void)
          {
            /* just draw the pixel red */
            gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
          }"
      }
    }
  }
  geometry Sphere { radius 2 }
}
```

A little longer example, showing X3D with the previous shader code (multiplying texture by 2):

```
#X3D V3.2 utf8
PROFILE Interchange
Shape {
  appearance Appearance {
    shaders ComposedShader {
      language "GLSL"
      inputOutput SFNode myTexture ImageTexture { url "test_texture.png" }
      parts [
        ShaderPart {
          type "VERTEX"
          url "data:text/plain,
            void main(void)
            {
              gl_TexCoord[0] = gl_MultiTexCoord0;
              gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
            }"
        }
        ShaderPart {
          type "FRAGMENT"
          url "data:text/plain,
            uniform sampler2D myTexture;
            void main(void)
            {
              gl_FragColor = texture2D(myTexture, gl_TexCoord[0].st) * 2.0;
            }"
        }
      ]
    }
  }
  geometry Sphere { radius 2 }
}
```

The shader code provided this way *replaces* the standard calculations done by the GPU. This means that all the lighting and texturing effects, if needed, have to be reimplemented from scratch in our shader. There is no way to combine our shader with standard rendering features and it is impossible to automatically combine two shader sources. This drawback reflects the design of the hardware. And the whole work presented in this paper strives to overcome this problem.

Chapter 4. Motivation and previous work

The popular real-time shading languages (OpenGL *GLSL* [GLSL], NVidia *Cg* [Cg] and Direct 3D *HLSL* [HLSL]) do not provide a ready solution for connecting shaders from independent sources. The `CgFX` and *HLSL* `.fx` files encapsulate shading language code in *techniques* (for various graphic card capabilities) and within a single technique specify operations for each rendering pass. In neither case can we simply connect multiple shader source code files and expect the result to be a valid program.

The X3D *Programmable shaders component* [X3D Shaders] makes the three shading languages mentioned above available to X3D authors. Complete shader code may be assigned to specific shapes. Although it does not offer any way of compositing shader effects, this component is still an important base for our work. It defines how to comfortably keep shader code inside X3D nodes. It also shows how to pass uniform values (including textures) to the shaders.

An old method to combine effects, used even before the shading languages, is a multi-pass rendering. Each rendering pass adds or multiplies to the buffer contents, adding a layer with desired effect. However, this is expensive ∉ in each pass we usually have to repeat some work, at least transforming and clipping the geometry. It is also not flexible ∉ we can only modify the complete result of the previous pass. In our work, we want to allow a single rendering pass to be as powerful as it can.

Arranging shader functions in a pipeline has similar disadvantages as multi-pass rendering, except there's no speed penalty in this case.

Common approach for writing a flexible shader code is to create a library of functions and allow the author to choose and compose them in a final shader to achieve the desired look. But this approach is very limited, as we cannot modify a particular calculation part without replicating the algorithm outside of this calculation. For example, if we want to scale the light contribution by a shadow function, we will have to also replicate the code iterating over the light sources.

Sh (http://libsh.org/, [Sh]) allows writing shader code (that can run on GPU) directly inside a C++ program. For this, Sh extends the C++ language (through C++ operator overloading and macros tricks). It allows an excellent integration between C++ code and shaders, hiding the ugly details of passing variables between normal code (that executes on CPU) and shader code (that usually executes on GPU). We can use object-oriented concepts to create a general shader that can later be extended, for example by overriding virtual methods. However, this is a solution closely coupled with C++. It's suitable if we have a 3D engine in C++, we want to use it in our own C++ program and extend its shaders. Our solution is simpler, treating shader effects as part of the 3D content and can be integrated into a renderer regardless of its programming language. We do not need a C++ compiler to generate a final GPU shader and users do not need to be familiar with C++.

OGRE (http://www.ogre3d.org/), an open-source 3D engine written in C++, has a system for adding shader extensions (see [OGRE Shader]). Its idea is similar to our system (enhance the built-in shaders with our own effects), however the whole job of combining a shader is done by operating on particular shader by C++ code. The developer has to code the logic deciding which shaders are extended and most of the specification about how the extension is called is done in the C++ code. This has the nice advantage of being able to encapsulate some fixed-function features as well, however the whole system must be carefully controlled by the C++ code. In our approach, we allow the authors to write direct shading language code quickly and the integration is built inside appropriate X3D nodes.

AnySL [AnySL] allows to integrate internal renderer shaders with user shaders, by introducing a new shader language.

Spark [Spark] is a recent work presenting a new language to develop composable shaders for GPU.

In this work, we deliberately decided not to introduce a new shading language. One of the problems with introducing a new language is that it is ~~yet~~ another language to learn~~y~~ for developers and users. For developers of 3D rendering applications, there's an additional effort of integrating the new language with a renderer, which often is not a trivial task. This creates a practical problem for new languages ~~¢~~ because they are not popular, it's even harder for them to become popular. Of course, a new language has also the possibility to introduce new features, and ~~win~~ developers this way. But we think that existing shading languages, like *GLSL*, are already comfortable for a lot of practical purposes. That's, in a nutshell, our motivation behind *extending* an existing shading language, instead of inventing a new one.

[X3D DeclarativeShader] presents a declarative approach to an advanced shader in X3D. However, it only allows a fixed set of functionality, kind of an *advanced and enhanced material*. It does not expose any shader functionality to the authors. It merely allows the authors to use some advanced algorithms that in practice will be usually implemented by shaders inside the application.

[X3D Volume] proposes a new X3D component for rendering volumetric data (coming from 3D textures). It is interesting in relation to our work, as it allows to compose various rendering styles for visualizing volumetric data. Many styles of rendering are predefined (all the nodes descending from the `X3DComposableVolumeRenderStyleNode`) and they can be combined together by the `ComposedVolumeStyle` node. The effects introduced in our work could serve as a low-level implementation for such rendering styles, utilizing GPU shading languages. Each rendering style could be defined as a prototype that expands into our `Effect` node, overriding some plug receiving data about the 3D volume, add adding the necessary visualization.

At the end, we would like to mention a solution from a completely different domain, that is surprisingly similar to ours in some ways. Drupal (http://drupal.org/), an open-source CMS system written in PHP, has a very nice system of modules. Each module can extend the functionality of the base system (or other module) by implementing a *hook*, which is just a normal PHP function with a special name and appropriate set of parameters. Modules can also define their own hooks (for use by other modules) and invoke them when appropriate. This creates a system where it's trivially easy to define new hooks and to use existing hooks. Many modules can implement the same hook and cooperate without any problems. The whole hook system is defined completely in PHP, as it's a scripting language and we can query the list of loaded functions by name, and call function by its name. Drupal approach is quite similar to our core idea of combining effects. Our effects are similar to Drupal's modules and our ~~plugs~~ are analogous to Drupal hooks.

Chapter 5. Extending the shaders with plugs

The core idea of our approach is that the base shader defines points where calls to user-defined functions may be inserted. We call these places *plugs*, as they act like sockets where logic may be added. Each plug has a name and a given set of parameters. The effects can use special function names, starting with `PLUG_` and followed by the plug name. These declarations will be found and the renderer will insert appropriate calls to them from the base shader.

A trivial example of an effect that makes colors two times brighter is below. This is a complete X3D file, so you can save it as `test.x3dv` and open with any tool supporting our extensions, like `view3d-scene`.

```
#X3D V3.2 utf8
PROFILE Interchange
Shape {
  appearance Appearance {
    material Material { }
    effects Effect {
      language "GLSL"
      parts EffectPart {
        type "FRAGMENT"
        url "data:text/plain,
        void PLUG_fragment_modify(
          inout vec4 fragment_color)
        {
          fragment_color.rgb *= 2.0;
        }"
      }
    }
  }
  geometry Sphere { }
}
```

Our extensions to X3D are marked with the bold font in the example above. The *GLSL* code inside our extensions is marked with the italic font. The special *GLSL* function name `PLUG_fragment_modify` indicates that we use the `fragment_modify` plug. This particular plug is called after calculating everything else for this pixel (textures, lighting) and allows to intuitively ¥modify the final pixel color¥. `fragment_color` is an ¥inout¥ parameter, by modifying it we modify the color that will be displayed on the screen.

A reference of all the plugs available in our implementation is at the end of this paper, see Appendix A, *Reference of available plugs*. For each plug, like this `PLUG_fragment_modify`, we define a list of parameters and when it is called.

Many usage scenarios are possible:

1. The `Effect` nodes may use plug names defined inside the renderer internal shaders. This is the most usual case. It allows the authors to extend or override a particular shading parameter.

2. The `Effect` nodes may also use the plug names defined in the previous `Effect` nodes on the same shape. It is trivially easy (just add a ¥magic¥ comment) to define plugs in your own shader code. This way your own effects can be customized.

3. Inside the renderer implementation, the same approach can be used to implement some internal effects. We have reimplemented many internal effects of our engine, like the fog, shadow maps (see our shadow mapping extensions for X3D [X3D Shadow Maps]) and the bump mapping to use our "plugs" approach. This made their implementation very clean, short and nicely separated from each other. It also proves that the authors have the power to implement similar effects easily by themselves.

Actually, there are even more possibilities. We have been talking above about the "renderer internal shaders", but the truth is a little more flexible. When you place a standard shader node (like a `ComposedShader` node for *GLSL* shaders) on the `Appearance.shaders` list, then it replaces the internal renderer shaders. If you define the same (or compatible) plugs inside your shader, then the internal renderer effects are even added to your own shader. Of course user effects are added to your shader too. This way even the standard X3D shader nodes become more flexible. Note that if you do not define any plugs inside your `ComposedShader` node, it continues to function as before – no effects will be added.

5.1. Effect node

New `Effect` node holds information about the source code and uniform values specific to a given effect. The node specification below follows the style of the X3D specification [X3D].

```
Effect : X3DChildNode

SFString [] language ""
  # Language like "GLSL", "CG", "HLSL".
  # This effect will be used
  # only when the base renderer shader
  # uses the same language.

SFBool [in,out] enabled TRUE
  # Easily turn on/off the effect.
  # You could also remove/add the node
  # from the scene, but often toggling
  # this field is easier for scripts.

MFNode [] parts [] # EffectPart
  # Source code of the effect.

# A number of uniform values may also be
# declared inside this node.
```

Inside the `Effect` node a number of uniform values may be defined, passing any X3D value to the shader. Examples include passing current world time or a particular texture to the shader. Uniform values are declared exactly like described in the standard X3D *Programmable shaders* component [X3D Shaders].

The effect source code is split into a number of parts:

```
EffectPart : X3DNode, X3DUrlObject

SFString [] type "VERTEX"
  # Like ShaderPart.type:
  # allowed values are
  # VERTEX | GEOMETRY | FRAGMENT.

MFString [in,out] url []
```

```
  # The source code, like ShaderPart.url.
  # May come from an external file (url),
  # or inline (following "data:text/plain,").
  # In XML encoding, may also be inlined in CDATA.
```

Inside the effect part source code, the functions that enhance standard shaders behavior are recognized by names starting with `PLUG_`. Of course other functions can also be defined and used. Uniform variables can be passed to the effect, also varying variables can be passed between the vertex and fragment parts, just like with standard shader nodes.

In a single `EffectPart` node, many `PLUG_` functions may be declared. However, all plug functions must be declared in the appropriate effect type. For example, the `fragment_modify` plug cannot be used within a `VERTEX` shader. If the effect requires some processing per-vertex and some per-fragment, it is necessary to use two `EffectPart` nodes, with different types. This allows to implement our system for shading languages with separate namespaces for vertex and fragment parts (like *GLSL*). A single part may declare many variables and functions, but it must be completely contained within a given shader type.

Note that it is completely reasonable to have an `EffectPart` node with source code that does not define any `PLUG_xxx` functions. Such `EffectPart` node may be useful for defining shading language utility functions, used by other effect parts.

For shading languages that have separate compilation units (like the *OpenGL Shading Language*) the implementation may choose to place each effect part in such separate unit. This forces the shader code to be cleaner, as you cannot use undeclared functions and variables from other parts. It also allows for cleaner error detection (parsing errors will be detected inside the given unit).

5.2. Effects for particular shapes

There are various places where an `Effect` node may be used. To apply an effect on a given shape, it can be placed on the new `Appearance.effects` list:

```
Appearance

MFNode [] effects [] # Effect
```

All the effects on this list (with suitable language) will be used. This allows authors to define a library of independent shader effects and then trivially pick desired effects for each particular shape. Simply placing two effects on the `Appearance.effects` list makes them cooperate correctly.

Figure 5.1. Toon and Fresnel effects combined

Note that all introduced nodes benefit from X3D mechanism to reuse the nodes by reference (the `DEF` and `USE` keywords). Reusing the `Effect` nodes is most natural and allows to combine existing effects in any desired way. Reusing the `EffectPart` nodes is also useful, when some effects would like to share a particular piece of code. For example, the same `EffectPart` node, with a library of useful shading language functions, may be used for many effects.

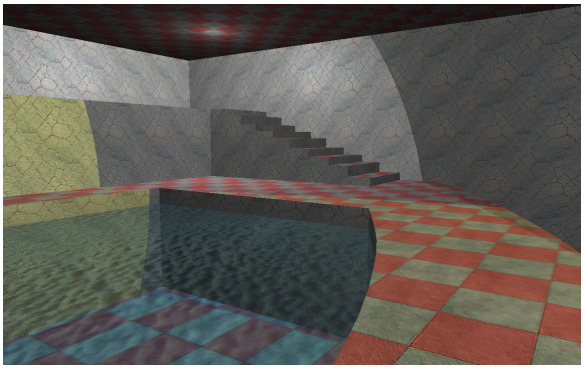
5.3. Effects for a group of nodes

The `Effect` node is a descendant of the abstract `X3DChildNode`. As such it can be placed directly within X3D grouping nodes like `Group`, `Transform` and at the top level of the X3D file. Such effect will apply to all the shapes within the given group. The scope rules follow the X3D conventions for other nodes, like pointing device sensor nodes and `LocalFog`.

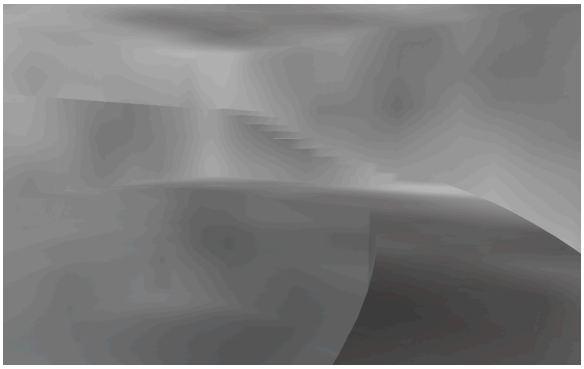
The `LocalFog` example is worth emphasizing. Using our system, an X3D viewer can implement the `LocalFog` node as a prototype that expands to our `Effect` node. This results in a 100% correct and easy implementation of the standard `LocalFog` node.

As one of the demos, we have implemented a realistic animated volumetric fog, where the fog density is stored in a 3D smooth noise texture (idea from [Volumetric Fog]). In a fragment shader, the 3D texture is sampled along the line between the camera and pixel position in the 3D space. This makes a very convincing effect of a dense fog. The `Effect` node with appropriate shader code is placed at the top level of the X3D file, so it simply works for all shapes. See Figure 5.2, ~~"~~Volumetric fog with animated density~~"~~.

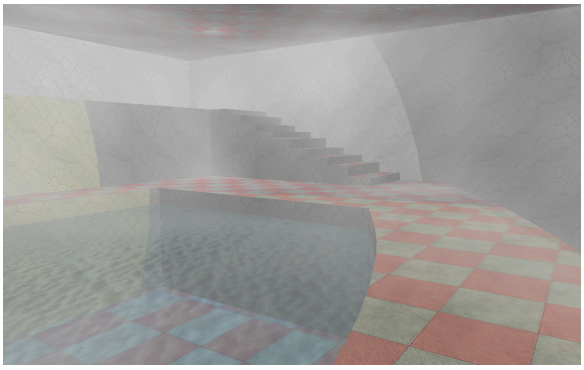
Figure 5.2. Volumetric fog with animated density



No fog.



No lighting. Note that the fog is assumed to have its own ambient lighting, so it colors the image even in this case.



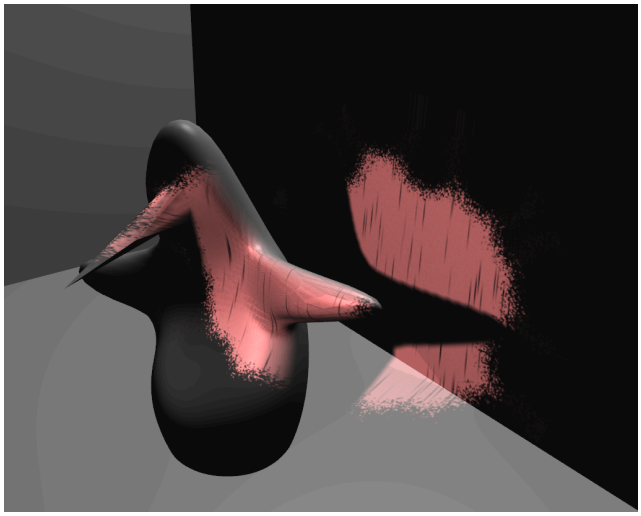
Lights and fog.

5.4. Light sources effects

The nice feature of our system is that effects can be attached to various types of objects, not just shapes. For example a particular light source may have a shader effect assigned. This allows to modify the contribution of a given light. For example the spot light shape can be modified, possibly based on some texture information (see Figure 5.3, ¥Textured spot light with shadow¥). Or a different lighting model may be implemented, like anisotropic Ward or Cook-Torrance. To make this possible, the `effects` field is added to every light node:

```
X3DLightNode

MFNode [] effects [] # Effect
```

Figure 5.3. Textured spot light with shadow



An example below demonstrates how light effects are specified inside an X3D. The code makes a spot light with an exponential drop-off, similar to traditional OpenGL fixed-function spot lights. Such code can replace (or multiply with) the standard X3D concept of the linear drop-off for spot lights.

The customization is done by overriding the `PLUG_light_scale`. Remember that all plugs available in our implementation are documented in Appendix A, *Reference of available plugs*.

```
SpotLight {
  effects Effect { language "GLSL"
    parts EffectPart { type "FRAGMENT"
      url "data:text/plain,
        uniform vec3 castle_LightSource0SpotDirection;

        void PLUG_light_scale(inout float light_scale,
          const in vec3 normal_eye,
          const in vec3 light_dir)
        {
          light_scale *= pow(dot(normalize(
            castle_LightSource0SpotDirection), -light_dir), 10.0) * 2.0;
        }"
    } } }
```

5.5. Texture effects

Just like the light sources, also each texture node may define its own effects:

```
X3DTextureNode

MFNode [] effects [] # Effect
```

`X3DTextureNode` is an ancestor for all the standard texture nodes, like the `ImageTexture`. This allows to modify any X3D texture by shader effects. A plug `texture_color` may be used to change the texture color, taking into account the current texture coordinates and other information.

5.5.1. Procedural textures

A new X3D node `ShaderTexture` is available for creating procedural textures using the GPU shading languages. The texture contents are not stored anywhere (not even on GPU) and the renderer does not manage any texture resources. From a GPU point of view, there is no texture. There is only a shader function that generates colors based on some vectors. By wrapping such function inside the `ShaderTexture` node, it can be treated exactly like other textures in the scene. In particular, texture coordinates (explicit or generated) can be comfortably provided for the procedural texture. Effectively, it behaves like a normal texture node, with all the related X3D features.

The new texture node specification:

```
ShaderTexture : X3DTextureNode

MFNode [] effects [] # Effect

SFString [] defaultTexCoord "BOUNDS2D"
  # ["BOUNDS2D"|"BOUNDS3D"]
```

Actually, the `effects` field is already defined in the base `X3DTextureNode` class mentioned previously. It is repeated here only for completeness.

An effect overriding the `texture_color` plug should be included, otherwise texture colors are undefined. Our implementation sets the default texture color to pink (*RGB(1, 0, 1)*), so it stands out, reminding author to override it.

The texture coordinates, or the algorithm to generate them, can be explicitly specified, just like for any other texture in X3D. This is done by placing any `X3DTextureCoordinateNode` node inside the geometry `texCoord` field. Both explicit texture coordinate lists (`TextureCoordinate`, `TextureCoordinate3D`, `TextureCoordinate4D`) as well as the coordinate generator nodes (like `TextureCoordinateGenerator` and `ProjectedTextureCoordinate`) are allowed. Note that projective texture mapping by the `ProjectedTextureCoordinate` is also our X3D extension, see [X3D Shadow Maps].

When the texture coordinates are not given, the `defaultTexCoord` field determines how they are generated:

1. `"BOUNDS2D"` generates 2D texture coordinates, adapting to the two largest bounding box sizes. The precise behavior of `"BOUNDS2D"` follows the X3D `IndexedFaceSet` specification.

   This is most comfortable when the texture color depends only on the XY components of the texture coordinate. The 3rd texture coordinate component is always 0, and the 4th component is always 1.

2. `"BOUNDS3D"` generates 3D texture coordinates. The texture coordinates are adapted to all three bounding box sizes, precisely following X3D specification section *"Texture coordinate generation for primitive objects"* of the *Texturing3D* component.

   This is most suitable for true 3D textures. The 4th texture coordinate component can be ignored. Or the 4D vector may be treated as homogeneous, as `"BOUNDS3D"` will always set the 4th component to 1.

The `"BOUNDS*"` names are consistent with another extension of our engine. We allow the same values to be used in the `TextureCoordinateGenerator.mode` field. See [X3D TexCoord Bounds].

In the end, the idea is that using a `ShaderTexture` should be as comfortable as any other texture node.

5.5.2. Example of a procedural texture

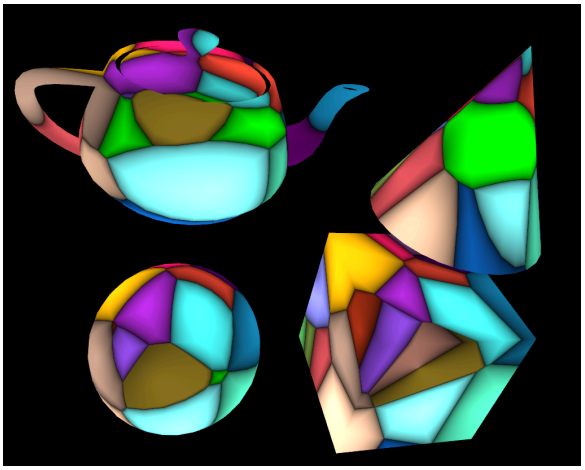
As an example, we present an outline of a procedural texture code using the *cellular texturing* idea. This is a nice approach to computing segmented textures, resembling various combinations of Voronoi diagrams. We override the `PLUG_texture_color` to calculate a *feature point* that is closest in 3D space to our current texture coordinate. The distance to this feature point, combined with the distance to the next-closest feature point, can be combined to achieve interesting visual effects.

```
ShaderTexture {
  effects Effect {
    language "GLSL"
    parts EffectPart {
      type "FRAGMENT"
      url "data:text/plain,
        #version 120

        void PLUG_texture_color(inout vec4 texture_color,
          const in vec4 tex_coord)
        {
          const int count = ...;
          const vec3 feature_points[count] = vec3[count](...);
          const vec3 feature_colors[count] = vec3[count](...);

          float[count] distances;
          int closest0, closest1;
          for (int i = 0; i < count; i ++)
          {
            distances[i] = distance(vec3(tex_coord), feature_points[i]);
            /* ...
             Update closest0 to indicate the closest feature point,
             that is distances[closest0] is smallest among all distances.
             Update closest1 to index of the 2nd-closest feature point.
            */
          }

          texture_color.rgb = pow(distances[closest1] -
            distances[closest0], 0.3) * 2.0 * feature_colors[closest0];
        }"
    }
  }
  defaultTexCoord "BOUNDS3D"
}
```

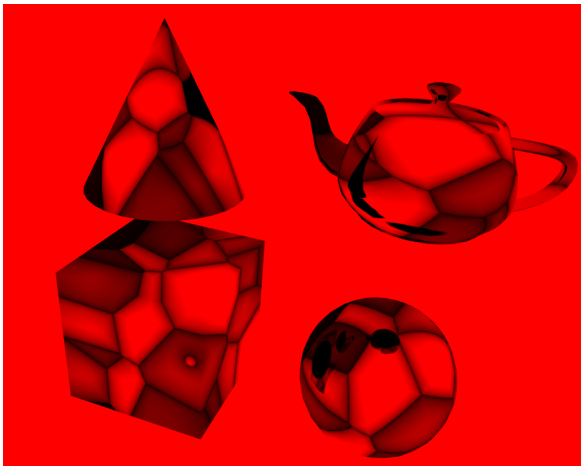
Note that this is a very simple approach to implementing *cellular texturing*. Much more optimal implementation is possible. There are also many variations that achieve different visual appeal. See our engine demo models (http://castle-engine.sourceforge.net/demo_models.php) for a complete and working implementation.

Figure 5.4, "Cellular texturing" shows this texture used in various settings. It can be combined with other textures without any effort — in this case, we show it combined with a cube map texture that simulates a mirror.

Figure 5.4. Cellular texturing



Cellular texturing.



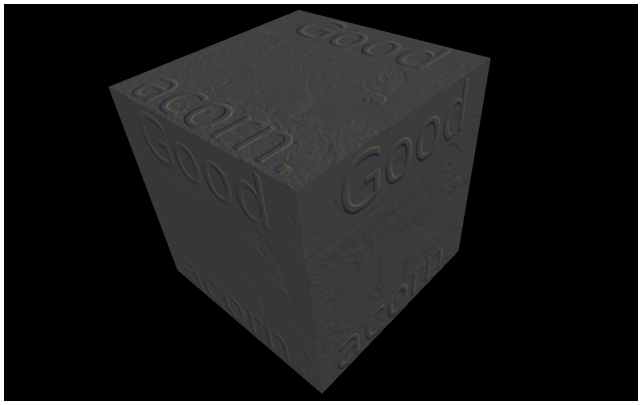
Cellular texturing mixed with a mirror by cube environment mapping.

5.5.3. When to use the ShaderTexture

For textures other than the `ShaderTexture`, when the `texture_color` plugs are called, the internal shaders have already calculated the initial texture color by actually sampling the texture image. This is useful if you want to modify this color. If you'd rather ignore the normal sampled color, and always override it with your own, consider using the special `ShaderTexture` node instead. Using a normal texture node (like `ImageTexture`) for this would be uncomfortable, as you would have to load a dummy texture image, and the shaders could (depending on optimization) waste some time on calculating a color that will be actually ignored later.

Note that in all cases (effects at `ImageTexture`, at `ShaderTexture`, etc.) you can always use additional textures inside the effect. Just like inside a standard `ComposedShader`, you can declare an `SFNode` field inside an `Effect` to pass any texture node to the shader as a uniform value. This allows to combine any number of textures inside an effect. The only difference between `ShaderTexture` and other textures is what the system does automatically for you, that is what color is passed to the first `texture_color` plug.

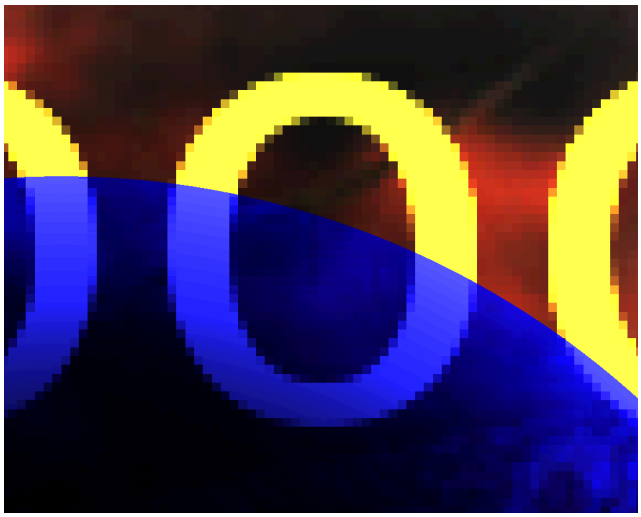
Figure 5.5. ShaderTexture doing an edge detection operation on a normal ImageTexture



5.5.4. Texture resolution does not matter

The shader effects for textures are calculated at each screen fragment, not at each texel. So the effects are not concerned with the texture size or texture filtering options. The `texture_color` plug receives the interpolated texture coordinates. Figure 5.6, “Texture effects are not concerned with the texture resolution” shows a blue arc drawn on a texture by our effect. The arc border is perfectly smooth, without any concern about the pixel resolution of the underlying texture.

Figure 5.6. Texture effects are not concerned with the texture resolution



Chapter 6. Extensions for geometry shaders

Geometry shaders are an optional stage of 3D rendering. A geometry shader program is executed once for each primitive, like a single triangle. It works between the *vertex* and *fragment shader* ¢ it knows all the outputs from the vertex shader, and is responsible for passing them to the rasterizer. Geometry shader can use the information about given primitive (vertex positions and attributes) and can generate other primitives. A single geometry shader may generate any number of primitives, so it is possible to ~~explode~~ a single input primitive into a number of others. Or we can entirely discard some primitives. The type of the primitive may be changed by the geometry shader ¢ for example, triangles may be converted into points or the other way around.

Figure 6.1. Converting mesh into a dense line and point sets by geometry shaders

More details about using the geometry shaders with X3D and our engine are on http://castle-engine.sourceforge.net/x3d_implementation_shaders.php#section_geometry. The examples there show how to use the standard `ComposedShader` node to define a geometry shader for the shape. Here, we investigate how our effects improve the geometry shaders.

6.1. Robust geometry shaders

When we write a geometry shader, we want to control the logic of the primitive generation and the output primitive type. This means that the shader author wants to provide the main part of the geometry shader, that contains the `main()` entry point and necessary `layout` declarations.

We want to enable integration of our effects with user geometry shaders. It must be possible to write a flexible geometry shader code, that works with any internal effects and user effects in `Effect` nodes. In other words, when writing the geometry shaders, we do not want to hardcode what values have to be passed from vertex processor to the rasterizer. To make this possible, with every geometry shader we will link additional code with three functions:

- `void geometryVertexSet(const int index)` ¢ set output vertex to be equal to the input vertex of the given index.
- `void geometryVertexZero()` ¢ set all output vertex attributes to zero. This is really useful only before doing a series of `geometryVertexAdd` calls.

- `void geometryVertexAdd(const int index, const float scale)` – add to the output vertex given input vertex, scaled.

The idea is that `geometryVertexSet` can be used to simply pass-through values from vertex processing to the rasterizer. Calling `geometryVertexSet(i)` is equivalent (but possibly more efficient) to

```
geometryVertexZero();
geometryVertexAdd(i, 1.0);
```

For example, this is a trivial pass-through geometry shader, that doesn't do anything. Thanks to using `geometryVertexSet`, every other effect still works (without `geometryVertexSet`, various effects would break, because values would not be passed from vertex shaders to fragment shaders). This is what we mean by "robust" geometry shaders.

```
effects Effect {
  language "GLSL"
  parts EffectPart {
    type "GEOMETRY"
    url "data:text/plain,
      #version 150

      layout(triangles) in;
      layout(triangle_strip, max_vertices = 3) out;

      void geometryVertexSet(const int index);

      void main()
      {
        for(int i = 0; i < gl_in.length(); i++)
        {
          gl_Position = gl_in[i].gl_Position;
          geometryVertexSet(i);
          EmitVertex();
        }
        EndPrimitive();
      }"
  }
}
```

For more elaborate cases, `geometryVertexAdd` may be used to blend many input vertexes into one final output vertex. An example below shows a geometry shader that replaces every triangle with an averaged single point.

```
effects Effect {
  language "GLSL"
  parts EffectPart {
    type "GEOMETRY"
    url "data:text/plain,
      #version 150

      layout(triangles) in;
      layout(points, max_vertices = 1) out;

      void geometryVertexZero();
      void geometryVertexAdd(const int index, const float scale);
```

```
      void main()
      {
        gl_Position = (
          gl_in[0].gl_Position +
          gl_in[1].gl_Position +
          gl_in[2].gl_Position ) / 3.0;
        geometryVertexZero();
        geometryVertexAdd(0, 1.0 / 3.0);
        geometryVertexAdd(1, 1.0 / 3.0);
        geometryVertexAdd(2, 1.0 / 3.0);
        EmitVertex();
        EndPrimitive();
      }"
  }
}
```

The `geometryVertexXxx` functions ~~magically~~ take into account all the attributes that need to be handled for renderer internal effects. User effects (`Effect` nodes) may have to override corresponding `geometry_vertex_xxx` plugs to also be automatically handled this way.

6.2. Effects that cooperate with geometry shaders

User effects that want to cooperate with geometry shaders have to override plugs `PLUG_geometry_vertex_xxx`. The exact specification of these plugs is in Section A.4, ~~"~~Geometry shader plugs~~"~~. Overriding these plugs allows to pass-through or blend custom vertex attributes needed by the effects. By correctly overriding them, the user effects can work regardless if the geometry shader is linked or not.

To override these plugs, user effects must include an `EffectPart` with `type` set to `"GEOMETRY"`. This `EffectPart` code is optional. It will be used if an other effect will provide a `main()` entry for geometry shaders, and discarded otherwise.

Our goal throughout this paper is to make effects independent from each other, and composable with each other. See the example `compositing_shaders/geometry_shader_optional.x3dv` in our demo models. It shows that all kinds of effects, including geometry shader effects, may be created and applied independently from each other. As always, just placing two or more effects together on the X3D `effects` field makes them automatically cooperate.

There is one obstacle here. In case of user-defined vertex attributes, using the geometry shader means that the attribute name must change on its way from the vertex shader to the fragment shader. That is because you have to use different input and output names for this attribute inside the geometry shader. On the other hand, when there is no geometry shader, attribute name must be exactly the same in both the vertex shader and fragment shader. This means that creating a vertex, fragment and geometry shader combo in which the geometry shader is optional is not possible in pure shading language like *GLSL*. To overcome this, we automatically define a symbol `HAS_GEOMETRY_SHADER` for all fragment shaders' parts. This way we can write in the fragment shader code like:

```
#ifdef HAS_GEOMETRY_SHADER
  #define my_attribute my_attribute_fragment
#endif
in float my_attribute;
```

Such fragment shader can receive its input either from the geometry shader (under the name `my_attribute_fragment`) or straight from the vertex shader (under the name `my_attribute`).

Chapter 7. Defining custom plugs

In a shader code, new plug may be defined by a magic comment:

```
/* PLUG: name (param1, param2, ...) */
```

This defines a point where calls to user functions declared as `PLUG_name` will be inserted. They will be called with given parameters.

Many effects may use the same `PLUG_name`. Even within a single effect, the same `PLUG_name` may be used many times. All the `PLUG_name` functions will be uniquely renamed to not collide with each other.

The calls will be added in the order they are specified on the `effects` list. More precisely, the most local effects (at light sources and textures) are called first, then the effects at shape appearance, and finally the effects inside the grouping nodes. Although, preferably, for most effects this order will not matter.

For the effects on lights and textures, we first try to find the plug specific to the given light or texture node. This means that using the `PLUG_light_scale` inside the `X3DLightNode.effects` changes only the given light node contribution. Contrast this with using the same `PLUG_light_scale` inside `Appearance.effects`, in which case the intensity of all the light sources on the given shape can be changed.

A plug is often defined to allow modifying some parameter repeatedly (like adding or modulating the fragment color), so one or more of the parameters are often allowed to be handled as "inout" values.

The same plug name may be defined many times in the source shader. That is, the magic comment `/* PLUG: name ... */` may be repeated a couple of times, with the same `name`. This means that the final shader may call every matching `PLUG_name` function many times. This is useful when the algorithm is naturally expressed as a loop, but it had to be unrolled for shader source (for example, to slightly tweak some loop iterations).

Currently all the plugs must be procedures, that is their result type must be declared as `void`. We have been considering a possibility of functions, where part of the calculation may be replaced by a call to a plugged function. While not difficult to implement, this idea seems unnecessary after many tests. Procedural plugs are easier to declare, as the call to the plug may be simply inserted, while in case of function it will have to replace some previous code. This also means that using a procedural plug *never* replaces or removes some existing code, which is a very nice concept to keep. We want the effects to cooperate with each other, not to "hijack" from each other some parts of the functionality.

New plugs can be defined inside the `Effect` nodes, as well as inside the complete shaders (like standard `ComposedShader` nodes). In the first case, the plugs are only available for the following effects of the same shape.

The advantage of using magic comments to define plugs is that they can be ignored and a shader source remains valid. This means that `ComposedShader` nodes can define custom plugs and still work (although with no extra effects) even in X3D browsers that do not support our extensions.

7.1. Forward declarations

Suppose we have an effect *X* that defines a new plug, by including a magic `/* PLUG: ... */` comment. When this plug is used by another effect *Y*, then an appropriate function call is automatically

inserted into the generated shader. In the middle of the source code of effect *X*, a function defined in effect *Y* has to be called. This is the simplest implementation of our plugs.

Additionally, a forward or external declaration of the called function may need to be inserted into the effect *X*. That is because *Y* may be in a separate compilation unit (in case of *GLSL*), or just defined lower in the code. In simple cases, such forward or external declarations can be inserted right at the beginning of effect *X* code.

Some shading language directives are required to be placed before all normal declarations. For example, in case of the *OpenGL shading language*, the `#version` as well as some `#extension` directives must occur at the beginning of the shader code. To handle such cases, another magic comment `/* PLUG-DECLARATIONS */` is available. If present, it signifies a place where forward or external declarations should be inserted.

7.2. Invalid shader code

The behavior is defined only if the provided shading language code is a correct, self-contained code. The errors (like unterminated block) may only be detected after the complete shader is determined and compiled by the GPU. It should be noted that for shading languages with separate compilation units, the parsing errors can be at least reported always for the correct code piece (effect part).

An invalid effect code may disable all other user effects on the given shape. That is because there's no reliable way to detect which user effect prevents the compilation. At least for shading languages without the *separate compilation units* feature. In such case, the application may decide to disable *all* user-provided effects for a given shape. However, this isn't exactly a new problem ¢ bad shader code may always cause enough trouble to prevent the shape from being sensibly rendered.

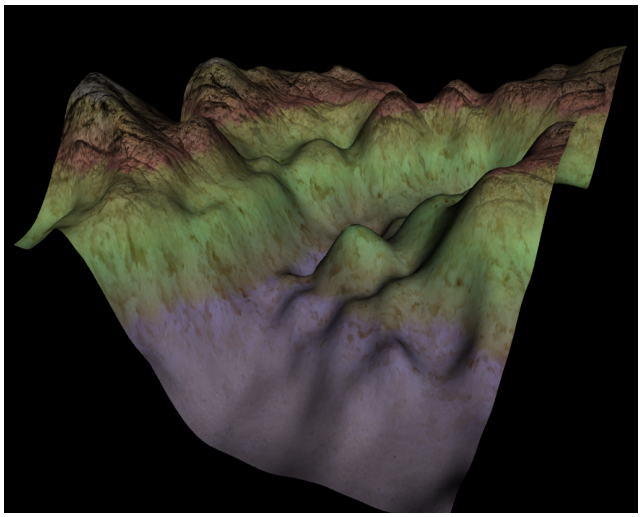
The **application does not need to parse the shader code** at any point. Only a trivial text search in the code is necessary to detect the magic plug function names and comments.

Chapter 8. Examples

A static screenshot will never express the freedom of movement in an animated 3D scene. So we encourage you to try the examples mentioned in this chapter yourself. Download `view3dscene`, our X3D browser, from http://castle-engine.sourceforge.net/view3dscene.php. Then download our demo models from http://castle-engine.sourceforge.net/demo_models.php. You can now run `view3dscene`, and open with it various models inside `demo_models/compositing_shaders/` subdirectory. Also the water demos inside `demo_models/water/` should be interesting.

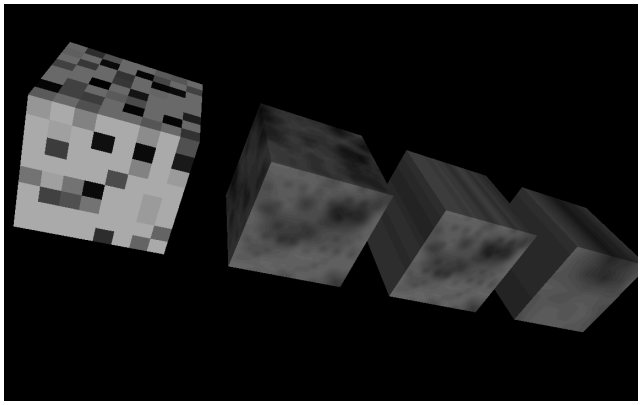
Effects may define and use their own uniform variables, including textures, just like the standard shader nodes. So we can combine any number of textures inside an effect. As an example we wrote a simple effect that mixes a couple of textures based on a terrain height. See Figure 8.1, "ElevationGrid with 3 textures mixed (based on the point height) inside the shader". We could also pass any other uniform value to the effect, for example passing the current time from an X3D `TimeSensor` allows to make animated effects.

Figure 8.1. ElevationGrid with 3 textures mixed (based on the point height) inside the shader



We can wrap 2D or 3D noise inside a `ShaderTexture`. See Figure 8.2, "3D and 2D smooth noise on GPU, wrapped in a ShaderTexture". A texture node like `NoiseTexture` from InstantReality [NoiseTexture] may be implemented on GPU by a simple prototype using the `ShaderTexture`.

Figure 8.2. 3D and 2D smooth noise on GPU, wrapped in a ShaderTexture

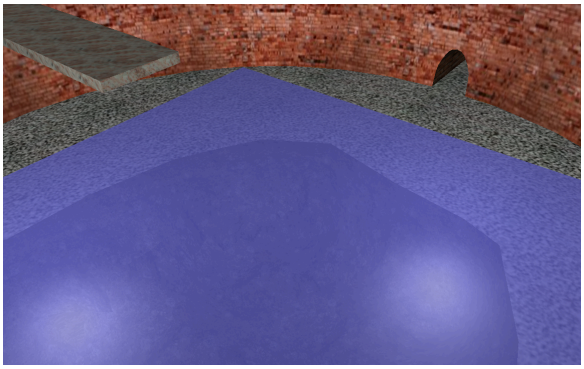


Water can be elegantly implemented using our effects, as a proper water simulation is naturally a combination of a couple effects. To simulate waves we want to vary vertex heights, or vary per-fragment normal vectors (for best results, we want to do both things). We also want to simulate the fact that water has reflections and is transparent. We have implemented a nice water using this approach, with (initially) two independent effect nodes. See Figure 8.3, ¥Water using our effects framework¥.

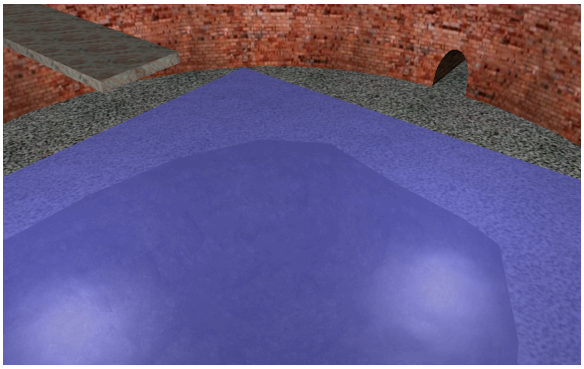
We were also able to easily test two alternative approaches for generating water normal vectors. One approach was to take normals from the pre-recorded sequence of images (encoded inside X3D `MovieTexture`, with noise images generated by the *Blender* renderer). The second approach was to calculate normals on the GPU from a generated smooth 3D noise. The implementation of these two approaches is contained in two separate `Effect` nodes, and is concerned only with calculating the normal vectors in the object space. Yet another `Effect` node is responsible for transforming these normal vectors into the eye space. This way we have extracted all the common logic into a separate effect, making it clear where the alternative versions differ and what they have in common. This was possible because one effect can define new plug names, that can be used by the other effects.

As for the question ¥Which approach to generating water normals turned out to be better?¥ Predictably, we showed that using GPU noise is slower, requires a better GPU, but also improves the quality noticeably. With GPU noise, there is no problem with aliasing of the noise texture and the noise parameters can be adjusted in real-time.

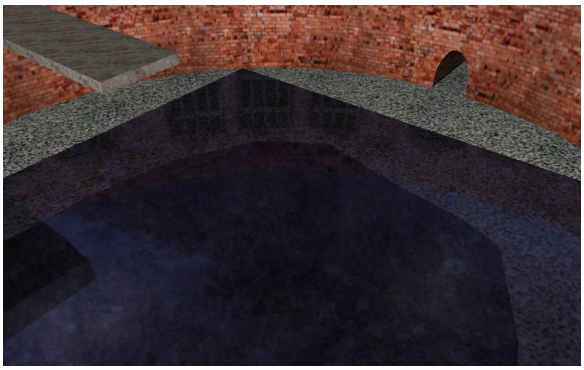
Figure 8.3. Water using our effects framework



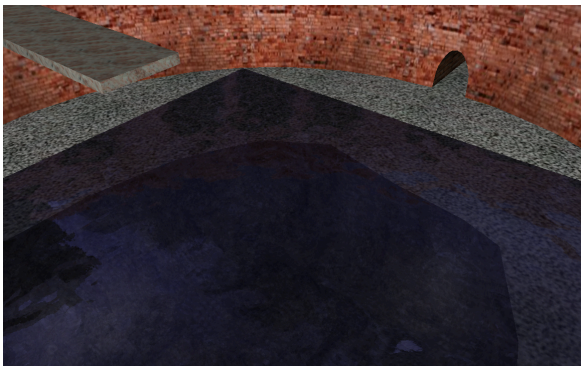
Per-pixel lighting.



Bump mapping.



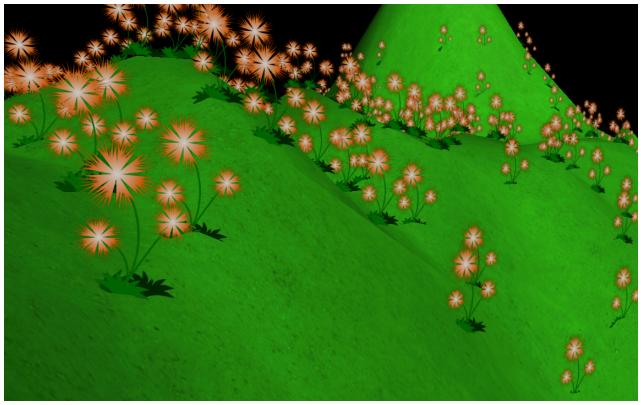
Reflections and refractions.



All effects.

We also have plugs to change the geometry in object space. Since the effect is automatically integrated with all the browser shaders, you only need to code a simple function to change the vertex positions. The effect instantly works with all the lighting and texturing conditions. Since the transformation is done on GPU, there's practically no speed penalty for animating thousands of flowers in our test scene. See Figure 8.4, ¥Flowers bending under the wind, transformed on GPU in object space¥.

Figure 8.4. Flowers bending under the wind, transformed on GPU in object space



We would like to emphasize that all the effects demonstrated here are theoretically already possible to implement using the standard X3D *Programmable shaders component* [X3D Shaders]. However, such implementation would be extremely cumbersome. You would first have to implement all the necessary multi-texturing, lighting, shadows, and other rendering features in a shader code. This is a large work if we consider all the X3D rendering options. Also note that a shader should remain optimized for a particular setting. The only manageable way to do this, that would work for all the lighting and texturing conditions, is to write a shader generator program. Which is actually exactly what our effects already do for you ¢ the implementation of our effects constructs and links the appropriate shader code, gathering the information from all the nodes that affect the given shape. The information is nicely integrated with X3D, effects are specified at suitable nodes, and their uniform values and attributes are integrated with X3D fields.

Chapter 9. Implementation

We have implemented everything described in this paper in our open-source (LGPL) 3D game engine, see [Castle Game Engine].

Our current implementation supports only one shading language: *GLSL*, the *OpenGL Shading Language*. As our engine is cross-platform and focused on OpenGL, this is the most natural shading language for us. However, we have designed our extensions to be applicable to other shading languages (like *Cg* and *HLSL*) as well.

One notable concept of *GLSL* is the *separate compilation units*. It means that a code can be split into many units which are parsed and compiled separately, and only linked together. This allows to write cleaner shader code (you cannot use undeclared functions from other shader parts). It also naturally matches with our effects definition, as each `EffectPart` becomes simply one compilation unit.

A similar feature is found in many other programming languages, under the names of "units" or "modules". But it is not available in two other popular *shading* languages: *Cg* and *HLSL*. To make sure that our idea is applicable to these shading languages, we have explicitly tested that even without the *separate compilation units*, our implementation still works without problems.

9.1. Outline of the implementation

We present a pseudo-code that generates a complete shader source for rendering a given shape. It takes into account standard rendering features (X3D light sources, textures and such), custom shaders (by X3D nodes like `ComposedShader`) and our shader effects (by `Effect` nodes). All effects are properly combined to form the final shader code.

We keep the final shader code as three arrays of strings. Each array keeps code for a specific shader type: vertex, geometry and fragment. Each string corresponds to a compilation unit for *GLSL*, for other shading languages the strings can be just concatenated at the end. To make the plugs actually work, we add calls to their functions. We also add *external function declarations* at appropriate places. For languages other than *GLSL*, they will simply become *forward function declarations* when all the parts are concatenated.

9.1.1. Helper functions

First define a function `Plug`. It is responsible for actual text processing that makes the plug functions correctly called. The argument `PlugValue` is scanned for all `PLUG_xxx` function definitions.

The argument `CompleteCode` is searched for the matching `/* PLUG: xxx ... */` comments. The final shader (for given shape) in `FinalShader` is also searched for the `/* PLUG: xxx ... */` comments. In practice, `CompleteCode` given here is either the `FinalShader` or the shader for a specific texture or light source.

Appropriate calls and forward declarations are inserted to the `CompleteCode`. In the process, all handled `PLUG_xxx` functions inside `PlugValue` are also renamed to unique names, since a single plug may be overridden by many effects. The modified `PlugValue` is inserted to the `CompleteCode` as well. Effectively, the caller can usually "forget" about the `PlugValue` afterwards — it has been processed, and correctly merged with the `CompleteCode`.

```
type
  TShaderType = (vertex, geometry, fragment);
  TShaderSource = array [TShaderType] of a string list;
```

```
var
  { shader for the whole shape }
  FinalShader: TShaderSource;

procedure Plug(
  EffectPartType: TShaderType;
  PlugValue: string;
  CompleteCode: TShaderSource);

var
  PlugName, ProcedureName, PlugForwardDeclaration: string;

  { Look for /* PLUG: PlugName (...) */ inside
    given CodeForPlugDeclaration.
    Return if any occurrence found. }
  function LookForPlugDeclaration(
    CodeForPlugDeclaration: string list): boolean;
  begin
    Result := false
    for each S: string in CodeForPlugDeclaration do
    begin
      AnyOccurrencesHere := false
      while we can find an occurrence
        of /* PLUG: PlugName (...) */ inside S do
      begin
        insert into S a call to ProcedureName,
        with parameters specified inside the /* PLUG: PlugName (...) */,
        right before the place where we found /* PLUG: PlugName (...) */

        AnyOccurrencesHere := true
        Result := true
      end

      if AnyOccurrencesHere then
        insert the PlugForwardDeclaration into S,
        at the place of /* PLUG-DECLARATIONS */ inside
        (or at the beginning, if no /* PLUG-DECLARATIONS */)
    end
  end

var
  Code: string list;
begin
  Code := CompleteCode[EffectPartType]

  HasGeometryMain := HasGeometryMain or
    ( EffectPartType = geometry and
      PlugValue contains 'main()' );

  while we can find PLUG_xxx inside PlugValue do
  begin
    PlugName := the plug name we found, the "xxx" inside PLUG_xxx
    PlugDeclaredParameters := parameters declared at PLUG_xxx function

    { Rename found PLUG_xxx to something unique. }
    ProcedureName := generate new unique procedure name,
```

```
    for example take 'plugged_' + some unique integer

    replace inside PlugValue all occurrences of 'PLUG_' + PlugName
    with ProcedureName

    PlugForwardDeclaration := 'void ' + ProcedureName +
    PlugDeclaredParameters + ';' + newline

    AnyOccurrences := LookForPlugDeclaration(Code)

    { If the plug declaration is not found in Code, then try to find it
      in the final shader. This happens if Code is special for given
      light/texture effect, but PLUG_xxx is not special to the
      light/texture effect (it is applicable to the whole shape as well).
      For example, using PLUG_vertex_object_space inside
      the X3DTextureNode.effects. }
    if not AnyOccurrences and
       Code <> Source[EffectPartType] then
      AnyOccurrences := LookForPlugDeclaration(Source[EffectPartType])

    if not AnyOccurrences then
      Warning('Plug name ' + PlugName + ' not declared')
  end

  { regardless if any (and how many) plug points were found,
    always insert PlugValue into Code. This way EffectPart with a library
    of utility functions (no PLUG_xxx inside) also works. }
  Code.Add(PlugValue)
end
```

Using the `Plug` function, we can create the `EnableEffects` function. It handles the `effects` list, correctly processing it and adding to the given `CompleteCode`..

```
procedure EnableEffects(
  Effects: list of Effect nodes;
  CompleteCode: TShaderSource);
begin
  for each Effect in Effects do
    if Effect.enabled and
       Effect.language matches renderer shader language then
      for each EffectPart in Effect.parts do
        Plug(EffectPart.type, GetUrl(EffectPart.url), CompleteCode)
end
```

9.1.2. Final algorithm

Using the above functions, we construct the final shader code for given shape. All the effects (including effects specific to lights and textures) are correctly applied by the algorithm below. Specific requirements of the *geometry shaders* are also taken into account.

```
FinalShader := new TShaderSource
set FinalShader to basic rendering code
HasGeometryMain := false
```

At the beginning, `FinalShader` it set to a simple code that renders 3D object with no lights, no textures and no effects. The code contains magic `/* PLUG: xxx ... */` comments, which will allow to enhance it in the following steps.

The real *GLSL* shader code used by our engine at this step may be found in our engine sources. See http://svn.code.sf.net/p/castle-engine/code/trunk/castle_game_engine/src/x3d/opengl/glsl/template.vs for the vertex shader, http://svn.code.sf.net/p/castle-engine/code/trunk/castle_game_engine/src/x3d/opengl/glsl/template.fs for the fragment shader and http://svn.code.sf.net/p/castle-engine/code/trunk/castle_game_engine/src/x3d/opengl/glsl/template.gs for the geometry shader.

Note that our default geometry shader contains only a set of utility functions, without a `main()` entry. It will be discarded later if no geometry shader code with a `main()` definition will be found in the user shaders. That is, if `HasGeometryMain` will remain `false`. See Chapter 6, *Extensions for geometry shaders* for reasons of this behavior.

```
if a complete custom shader code is provided then
  FinalShader[fragment] := empty
  FinalShader[vertex] := empty
  FinalShader := FinalShader + custom shader code
  HasGeometryMain := custom shader code contains some "GEOMETRY" shader
```

The X3D file may contain shader code that should replace the default shaders, following [X3D Shaders] specification. For example, a `ComposedShader` node may be present with a complete *GLSL* code. We use it at this step.

This step trivially allows the `ComposedShader` code to also contain plug declarations, like `/* PLUG: xxx ... */`. The same plug names as our default names may be used (like `fragment_modify` and so on), in which case the same user effects will be useful with the custom shader. This even allows the browser to add some internal effects (like shadow maps) to the custom shader template.

Alternatively, the `ComposedShader` may have a completely different approach to rendering. Then it may expose a completely different set of plug names, reflecting a different set of parameters to control.

Note again the special treatment of geometry shaders. Our default geometry shader code (which should contain our utility functions) is always kept. We also update `HasGeometryMain`.

```
for each Light in shape.Lights do
  LightShader := new TShaderSource
  ❶ set LightShader to basic lighting code (optimized for this Light)
  ❷ EnableEffects(Light.Effects, LightShader)
  ❸ LightContribution := LightShader.ExtractFirst
  ❹ Plug(fragment, LightContribution, FinalShader)
  ❺ FinalShader := FinalShader + LightShader
```

A little care is needed to correctly add light sources. Remember that lights may have user-defined effects that should be applied only to the specific light source. That's why we temporarily keep the shader code specific to a given light source in a separate `LightShader` variable.

❶ The light source contribution will be linked with the final shader also using our plugs. We initially add to `LightShader` a function called `PLUG_add_light_contribution_side` that takes care of calculating light contribution, following normal X3D light equations. This function should be optimized for the given light type (spot, directional, point) and light parameters (for example, lights without an attenuation factor or an infinite radius or zero specular term may be optimized at this point). If we want *Phong shading*, this function should be added as the first string of

the `LightShader[fragment]`. If we want *Gouraud shading*, it should go to `LightShader[vertex]` instead.

The actual initial *GLSL* light shader code used by our engine at this step may be found in our engine sources, see http://svn.code.sf.net/p/castle-engine/code/trunk/castle_game_engine/src/x3d/opengl/glsl/template_add_light.glsl.

❷ Next we apply `Effect` nodes specific to this light source. After this, `LightShader` contains the initial code merged with user effects. Doing it this way means that the standard light source plugs (like `light_scale`) as well as custom plugs (defined in one `Effect` node and used by following `Effect` nodes) work correctly.

❸ After applying user effects, we extract (get and delete) from `LightShader` our initial code. It may be modified now, since calls to user effects are now present inside.

❹ The extracted `LightContribution` must now be connected with the `FinalShader` code. This can be done by a simple call to the `Plug` function, which will notice the `PLUG_add_light_contribution_side` present inside `LightContribution`. After this operation, everything is connected: final shader calls `PLUG_add_light_contribution_side`, which in turn calls user effects on the light source.

❺ Finally, add the remaining code to be linked together with the `FinalShader`. This step simply adds the strings from one list to the other, with no processing.

```
for each Texture in shape.Textures do
  TextureShader := new TShaderSource
  set TextureShader to basic code (optimized for this Texture)
  EnableEffects(Texture.Effects, TextureShader)
  TextureApplication := TextureShader.ExtractFirst
  Plug(fragment, TextureApplication, FinalShader)
  FinalShader := FinalShader + TextureShader
```

Texture effects require a similar approach as light effects, to correctly catch all the ways how plugs may be used.

We start by creating a default shader code that applies the texture, knowing the texture type (2D, 3D, cube map), texture mode (multiply, add and such) and other properties. It should follow all X3D texturing and multi-texturing requirements. The code should define a function named `PLUG_main_texture_apply` that can be later connected to the final shader. It should also declare plug named `texture_color`, that can be used by user effects for this texture.

There are actually more differences between the application of the light and texture effects. Section 9.2, ~~"~~Correct shadows from multiple light sources~~"~~ describes one feature that makes their logic slightly more complicated.

```
EnableEffects(appearance node.Effects, FinalShader)

for each group node containing this shape do
  EnableEffects(group node.Effects, FinalShader)
```

Effects specific to a given shape, and effects for all groups containing this shape, are applied.

The effects at this point may override also lights and textures plugs, like `light_scale`. That's simply because we have already added all the lighting and texturing shading code to the `FinalShader`.

Overriding `light_scale` at this point means that we scale the contribution of every light by the same function.

```
if HasGeometryMain then
  for each FragmentPart in FinalShader[fragment] do
    FragmentPart := '#define HAS_GEOMETRY_SHADER' + FragmentPart else
  FinalShader[geometry] := empty
```

Decide if we really want to link geometry shaders, based on whether we have `main()` for geometry shaders. If yes, then `HAS_GEOMETRY_SHADER` symbol has to be defined, as it may be useful for fragment shader authors. Otherwise, discard all geometry shader code.

At the end, `FinalShader` is just a collection of strings forming a shading language source code. For *GLSL*, each string is naturally a separate compilation unit, and can be compiled in isolation. For other shading languages, the parts may be simply concatenated together as necessary.

The full, actual source code of this operation is available in our engine sources, see the unit GLRendererShader. Source code is on http://svn.code.sf.net/p/castle-engine/code/trunk/castle_game_engine/src/x3d/opengl/glrenderershader.pas.

9.2. Correct shadows from multiple light sources

A texture may be a shadow map projected from a light source. In our paper [X3D Shadow Maps] we have noted that the shadow should scale only the appropriate light source contribution. This allows to observe correct shadows from multiple light sources. This means that shadow map textures must be used in a different stage of the calculation than normal textures.

Our plugs idea allows to do this, in a clean and concise way. At the place where `TextureShader` is created in the pseudo-code from the previous section, we treat shadow maps specially. If the light source corresponding to the shadow map affects our shape then we do not apply the texture in a usual way. Instead, we call the `Plug` function to augment the specific light source shader with a shadow check, like this:

```
if Texture is GeneratedShadowMap and
   Texture.light affects the shape then
  Plug(fragment,
    'uniform sampler2DShadow shadowMap01;
     void PLUG_light_scale(inout float scale, ...)
     {
       scale *= shadow2DProj(shadowMap01, shadowMap01TexCoord).r;
     }', LightShader);
```

The simple call to `shadow2DProj` may be replaced with a variant of the *Percentage Closer Filtering* (see [GPU Gems PCF]).

The calculation of light effects has to be complicated a little to make it work. In our simple pseudo-code, we added the effects to the `LightShader` and then we immediately merged `LightShader` with the `FinalShader`. Now, we need to remember the `LightShader` value for a longer time, in case it should be augmented with a shadow map. Alternatively, we could search for corresponding shadow maps at the moment when `LightShader` is created.

9.3. Pool of shaders

Straightforward use of the pseudo-code above means that we create new shader for each rendered shape. This works correctly, but is very time and memory consuming for large scenes. A good implementation should try to reuse the shaders. This can be achieved by keeping a *pool of available shaders*. For each newly created shader, we calculate a hash value (reflecting the whole shader configuration) and insert this shader into the pool. When a new shader is needed, we first look for it in the pool (using the hash value of the desired configuration), and if we find it — we reuse it (increasing the reference count). Only if the shader is not found, we create a new one. This is quite simple to do, and it provides a perfect sharing of shaders.

The hash value could be calculated based solely on the final string of the shader. However, this is too slow in our experience — as it means that all the string operations have to be performed before we even know the hash. It's much faster to calculate the hash value looking at all the parameters that will affect the generated shader source, including all the participating `Effect` and `ComposedShader` nodes, as well as the standard X3D lights sources, textures, fog parameters and so on. Only when we really need to create a new shader, then we calculate the actual shader source code and compile it. This means that our pseudo-code gets a little more complicated: most operations are in fact delayed. For example, at the first stage we only iterate over the light sources to update the hash value and remember the light source parameters. Later, if the actual source code is needed, we actually construct the shader code using the remembered light source parameters.

An additional advantage of the "pool of shaders" appears when a subset of the needed shaders is known in advance. For example, imagine an evening outdoor scene, with a storm in the distance. The lighting cracks the sky occasionally, making everything temporarily bright. In technical words, the scene has highly dynamic lighting, and the shaders for various lighting conditions must be swapped instantly, to keep the simulation smooth. In such case, an application may initialize the pool to contain some shaders with artificial non-zero use count. This way, some shader configurations are always kept initialized and ready to be used immediately.

9.4. Speed

Very nice thing about our effects framework is that it does not cause any speed loss. Effects code is just combined into the final shader code, without any transformations that could make it slower. Our process of "combining" effects is essentially adding function calls around. Fortunately, a function call has no speed penalty. Existing shading languages are defined such that functions can always be inlined (there is no recursion allowed, and parameter qualifiers have simple interpretation), and as far as we know they are actually always inlined by existing shading language compilers.

9.5. Inspect shaders generated by our implementation

You can run our `view3dscene` with `--debug-log-shaders` command-line option. Output will show you the final shader code generated, and also the OpenGL log after linking the shaders. Be sure to redirect the output to a file as it may be quite large. This is a useful way to learn about our shader rendering internals.

Another useful option to try in `view3dscene` is to switch to *View* → *Shaders* → *Enable For Everything* mode. This will force shader rendering for all the shapes, while by default we use shader rendering only for the shapes that require particular effects (shaders by `ComposedShader`, effects described in this

paper, shadow maps and such). Forcing shader rendering for everything allows to see how our shaders implement the whole X3D lighting and texturing model. It also forces all the lighting calculation to be done per-pixel, resulting in perfect specular highlights and spot light shapes.

Chapter 10. Conclusion

We show a new approach for developing effects using the GPU shading languages. It allows to combine various shader effects with each other and with application internal shaders. Our approach is relatively easy to implement and allows the authors to directly use the existing GPU shading languages. We propose a number of extensions to the X3D, an open standard for 3D data, to make our effects available for 3D content authors. We have implemented our approach for the *GLSL* shading language.

Appendix A. Reference of available plugs

Below is a quick reference of plugs available in our implementation. We have found these plugs to be sufficient for a wide range of effects, although of course there's always a place for changes and improvements. Remember that you can always define your own plugs in your effects and shader nodes.

Parameter names are shown below merely to document the parameter meaning. Of course you can change the parameter names when declaring your own plug function. To some extent you can also change the parameter qualifiers:

- If a parameter below is "inout", you can change it to "in", or "const in" if you don't want to modify the given value.
- You can also change the "inout" parameter to just "out", if you want to unconditionally overwrite the given value. Although this is usually not advised, as it means that you disable previous effects working on this parameter. Most of the time, summing or multiplying to the previous value is a better choice.
- If a parameter below is shown as "in", you can add or remove the *const* qualifier as you wish. Using *const* may allow the shader compiler for additional optimizations.

A.1. Vertex shader plugs

```
void PLUG_vertex_object_space_change(
  inout vec4 vertex_object,
  inout vec3 normal_object)
```

You can modify the vertex position and normal vector in object space here. If you don't need to modify the vertex position, consider using the `vertex_object_space` instead, that may result in more optimized shader.

```
void PLUG_vertex_object_space(
  const in vec4 vertex_object,
     inout vec3 normal_object)
```

Process the vertex and normal in object space. You cannot change the vertex position here, but you can still change the normal vector.

```
void PLUG_vertex_eye_space(
  const in vec4 vertex_eye,
  const in vec3 normal_eye)
```

Process the vertex and normal in eye (camera) space.

A.2. Fragment shader plugs

```
void PLUG_fragment_eye_space(
  const vec4 vertex_eye,
  inout vec3 normal_eye)
```

Process the vertex and normal in eye space, at the fragment shader. You can modify the normal vector here, this is useful for bump mapping.

Note that if you modify here normal vector, you may have to take care to properly negate it. When `gl_FrontFacing` is false, we're looking at the other side than where standard `gl_Normal` was pointing. For example, for bump mapping, it's most sensible to negate only the Z component of the normal vector in tangent space.

Note that this "plug" exists only when using *Phong shading*, not *Gouraud shading*.

```
void PLUG_texture_color(
  inout vec4 texture_color,
  [const in samplerXxx texture,]
  const in vec4 tex_coord)
```

Calculate or modify the texture color. This plug is available for texture effects. The second parameter is special: for `ShaderTexture`, it doesn't exist at all. For other texture nodes, the sampler type depends on the corresponding X3D texture node: `sampler2D` for 2D textures, `sampler3D` for 3D textures, `samplerCube` for cube maps, and `sampler2DShadow` for `GeneratedShadowMap`.

```
void PLUG_main_texture_apply(
  inout vec4 fragment_color,
  const in vec3 normal_eye)
```

Called right after *main texture* was applied. The main texture is

- `diffuseTexture` in case of Phong `Material`
- `emissiveTexture` in case of `UnlitMaterial`
- `baseTexture` in case of `PhysicalMaterial`

This plug is called always, even if the main texture is not actually present. You can change the fragment color now, for various effects.

There's a big difference between how Phong and Gouraud shading interact with this plug:

- In *Phong shading*, this plug is called *before* the lighting was applied. Because for Phong shading, the texture is the input for the lighting equation parameter.
- In *Gouraud shading* this plug is called *after* the lighting was applied. Because for Gouraud shading, the texture is applied in the fragment shader, and it is mixed with the color calculated from lights in the vertex shader.

```
void PLUG_texture_apply(
  inout vec4 fragment_color,
  const in vec3 normal_eye)
```

Deprecated name for `PLUG_main_texture_apply`. Do not use in new code.

```
void PLUG_fragment_modify(
  inout vec4 fragment_color)
```

Called *after lighting (including shadows) and textures are applied* (regardless of whether Phong or Gouraud shading is used). But *before gamma correction (to change color from linear to monitor col-*

or-space) and tone mapping are done. This is probably the most useful plug to intuitively *"change the fragment color"*.

```
void PLUG_fog_apply(
  inout vec4 fragment_color,
  const vec3 normal_eye_fragment)
```

At this point, the fog is applied. Again you can change here the fragment color, as you desire. This is called after lighting, textures and standard fog are all applied. You can use this to apply custom fog equation.

This happens *after gamma correction (to change color from linear to monitor color-space) and tone mapping are done* (because fog is done in final color space, to have fog color easily matching background and UI, see https://castle-engine.io/fog).

```
void PLUG_fragment_end(
  const in vec4 fragment_color)
```

Do the final processing of the fragment. This is called after applying both textures and fog, and *cannot modify the fragment color anymore*. This is useful for operations like alpha-testing the fragment.

A.3. Lights plugs (internal; at vertex or fragment shader)

The plugs in this section are available at the same shader stage where the lighting is calculated.

- For *Phong shading* they are available at the *fragment stage*.

  Phong shading is default since *Castle Game Engine* 7.0-alpha-snapshot on 2022-04-01, see https://castle-engine.io/wp/2022/04/01/design-lights-using-castle-game-engine-new-light-components-and-related-features-with-video/.

- For *Gouraud shading* they are available at the *vertex stage*.

We have not yet devised a portable way to specify them, to work regardless of the shading method used. A simple solution would be to allow a stage name like `"LIGHT"` that is an alias for `"FRAGMENT"` or `"VERTEX"`, depending on the current shading model (that may change for each `Shape` node).

Please treat these plugs as **internal** for now. We may break the compatibility of these plugs!

```
void PLUG_light_scale(
  inout float light_scale,
  const in vec3 normal_eye,
  const in vec3 light_dir)
```

Scale the light source contribution. This plug is available at light source nodes' effects, to scale a particular light. It can also be used in shape or group effects, in which case it will affect the contribution of all the lights on given shape.

The `light_dir` vector is a normalized direction to the light, in eye space.

```
void PLUG_material_light_ambient(
```

```
  inout vec4 ambient)
```

Ambient color may be changed here. The initial value is a multiplication of material and light ambient colors.

```
void PLUG_material_light_diffuse(
  inout vec4 diffuse,
  const in vec4 vertex_eye,
  const in vec3 normal_eye)
```

Diffuse color may be changed here. This is usually a multiplication of material and light diffuse colors, but you can change it here into anything you like.

```
void PLUG_material_light_specular(
  inout vec4 specular)
```

Specular color may be changed here. The initial value is a multiplication of material and light specular colors.

```
void PLUG_material_shininess(
  inout float shininess)
```

Shininess exponent may be changed here. The initial value is the material shininess exponent.

Note: the X3D `Material.shininess` and `CommonSurfaceShader.shininessFactor` are usually in a [0..1] range, and they are multiplied by 128.0 to calculate the actual exponent for light equations. This plug works with the actual exponent.

```
void PLUG_add_light_contribution(
  inout vec4 color,
  const in vec4 vertex_eye,
  const in vec3 normal_eye,
  in float material_shininess,
  in vec4 color_per_vertex)
```

Add color coming from lighting this material. This is used internally to add the light sources, with each light source adding another `add_light_contribution` plug.

```
void PLUG_lighting_apply(
  inout vec4 fragment_color,
  const vec4 vertex_eye,
  const vec3 normal_eye_fragment)
```

At this point, the lighting is calculated. Light contributions are summed, along with material emissive and global scene ambient colors, result is clamped to 1.0, and the alpha value is set correctly.

There's a big difference between how Phong and Gouraud shading interact with this plug:

- In *Phong shading*, this plug is called *in fragment stage, after both lighting and textures are applied*. Because for Phong shading, the lighting is calculated after textures.

- In *Gouraud shading* this plug is called *in vertex stage, after lighting but before texture application*. Because for Gouraud shading, the texture is applied later, in fragment shader.

A.4. Geometry shader plugs

Each geometry shader plug `PLUG_geometry_vertex_xxx` enhances what happens when the corresponding `geometryVertexXxx` function is called. See Chapter 6, *Extensions for geometry shaders* for details.

```
void PLUG_geometry_vertex_set(
  const in int index)
```

Set current geometry shader output to be equal to geometry shader input with given `index`. If your effect defines a custom varying value (output from vertex shader, input to fragment shader) then you should override this plug, to make geometry shaders working seamlessly with your effect.

```
void PLUG_geometry_vertex_zero()
```

Set current geometry shader output to be zero.

```
void PLUG_geometry_vertex_add(
  const in int index,
  const in float scale)
```

Add to the current geometry shader output value from input `index`, scaled by given `scale`.

References

[AnySL] Ralf Karrenberg, Dmitri Rubinstein, Philipp Slusallek and Sebastian Hack. "AnySL: efficient and portable shading for ray tracing". *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010. See http://portal.acm.org/citation.cfm?id=1921479.1921495. See also AnySL website http://www.cdl.uni-saarland.de/projects/anysl/.

[Castle Game Engine] Michalis Kamburelis. The engine webpage, with information and downloads of our tools (like `view3dscene`): http://castle-engine.sourceforge.net/.

[Cg] *Cg - The Language for High-Performance Realtime Graphics*. NVidia. See http://developer.nvidia.com/cg-toolkit.

[GLSL] Khronos Group. *OpenGL Shading Language*. See http://www.opengl.org/documentation/glsl/.

[GLSL Book] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2004.

[GPU Gems PCF] Michael Bunnell and Fabio Pellacini. "Shadow Map Antialiasing". Chapter 11 of: *GPU Gems*. 2004. Available online on http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html.

[HLSL] Microsoft. *HLSL*. See http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx.

[NoiseTexture] Instant Reality. *NoiseTexture*. See http://doc.instantreality.org/documentation/nodetype/NoiseTexture/.

[OGRE Shader] *OGRE Wiki - RT Shader System*. See http://www.ogre3d.org/tikiwiki/RT+Shader+System&structure=Development.

[Sh] Stefanus Du Toit and Michael McCool. *Metaprogramming GPUs with Sh*. A K Peters/CRC Press, 2004.

[Spark] Tim Foley and Pat Hanrahan. "Spark: Modular, Composable Shaders for Graphics Hardware". *Proceedings of SIGGRAPH 2011*. ACM, 2011. See http://graphics.stanford.edu/papers/spark/.

[Volumetric Fog] Emil Persson "Humus". *Volumetric Fogging 2*. 2006. See http://www.humus.name/index.php?page=3D&ID=70. Nice overview also on http://www.evl.uic.edu/sjames/cs525/shader.html.

[X3D] Web3D Consortium. *Extensible 3D (X3D) Graphics Standard*. 2008. ISO/IEC 19775-1.2:2008. See http://web3d.org/x3d/specifications/.

[X3D Bump Mapping] Michalis Kamburelis. *Bump mapping extensions*. 2008. See http://castle-engine.sourceforge.net/x3d_extensions.php#section_ext_bump_mapping

[X3D DeclarativeShader] Karsten Schwenk, Yvonne Jung, Johannes Behr and Dieter W. Fellner. "A modern declarative surface shader for X3D". *Proceedings of the 15th International Conference on Web 3D Technology*. ACM, 2010. See http://doi.acm.org/10.1145/1836049.1836051.

[X3D Shaders] Gonåalo Nuno Moutinho de Carvalho, Tony Gill and Tony Parisi. "X3D programmable shaders". *Proceedings of the ninth international conference on 3D Web technology*. ACM, 2004. Available online as part of the X3D specification, see http://web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/Part01/components/shaders.html

[X3D Shadow Maps] Michalis Kamburelis. *Shadow maps and projective texturing in X3D*. *Proceedings of the 15th International Conference on Web 3D Technology*. ACM, 2010. See http://castle-engine.sourceforge.net/x3d_extensions_shadow_maps.php.

[X3D TexCoord Bounds] Michalis Kamburelis. *Tex coord generation dependent on bounding box*. 2010. See http://castle-engine.sourceforge.net/x3d_extensions.php#section_ext_tex_coord_bounds.

[X3D Volume] Web3D Consortium. *Volume rendering component*. Part of the latest (not finalized yet) version of the *Extensible 3D (X3D) Graphics Standard*: X3D 3.3. 2011. The text of the component is inside the proposed X3D 3.3 specification: http://web3d.org/x3d/specifications/ISO_IEC_PDAM1_19775_1_2008-X3D-Abstract-Specification/Part01/components/volume.html.