

Compositing shaders in X3D

Michalis Kamburelis*

Institute of Computer Science
University of Wrocław, Poland



Abstract

We present a number of X3D extensions for implementing effects using the GPU shading languages. Our approach makes the effects naturally integrated with appropriate X3D nodes, like shape's appearance, light sources and textures. Our effects seamlessly cooperate with each other and with the shaders used internally by the X3D browser. Thus the effects are reusable, work in various combinations and under all lighting and texture conditions. This makes the GPU shaders useful for typical authoring needs. The browser implementation may also gain from using the same approach for composing internal shaders.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages, Standards

Keywords: X3D graphics, shaders, GLSL, shadows, shadow maps, bump mapping

1 Introduction

X3D [Web 3D Consortium 2008] is an open standard for representing rich 3D data. Many advanced graphic features are available to the authors. In particular, we can use *shaders* to change the look of the rendered geometry. *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 X3D standard *Programmable shaders component* already makes the shaders available to the authors. However, the shaders designed using the standard nodes *replace* the normal rendering functionality, not *enhance* it. This reflects the underlying API (like OpenGL or Direct 3D), and the idea that a 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 on top of it. The browser implementation usually already has an extensive internal shaders system, and the authors should be able to depend on these internal shaders to do the common job.

As an example, consider the simplified lighting equation with shadows and bump mapping:

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

The *shadow* function returns values in the [0..1] range, scaling the light color. If the object is not in the shadow, it returns 1. The *normal* function returns a normal vector at given point. Now we would like to configure various parts of this equation, without touching the others. For example, the *shadow* 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. The *light_color* function may be replaced to use different lighting models (Phong, Ward, Cook-Torrance and so on). Sometimes it makes sense to change these functions only for some chosen light sources, and sometimes we want to modify all the light sources.

We present a system for creating effects by essentially compositing together a pieces of shader code. All the effects defined this way effortlessly cooperate and can be combined with each other and with browser 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, and we can completely ignore other parts 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.

*e-mail: michalis.kambi@gmail.com

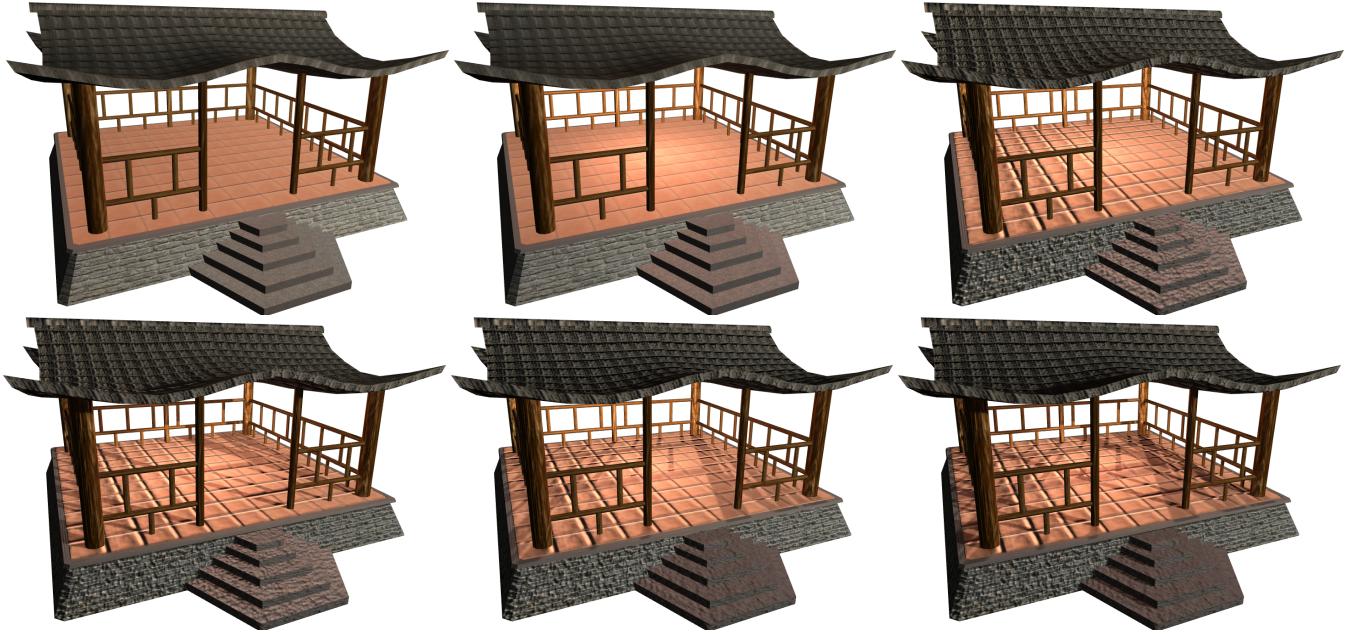


Figure 1: Japanese shrine model with more and more effects applied: Gouraud shading, Phong shading (per-pixel lighting), bump mapping, shadows from 1st light, shadows from 2nd light, shadows from both lights.

It is important that we still 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 X3D browser.

2 Motivation and previous work

The popular real-time shading languages (OpenGL GLSL, NVidia Cg, Direct 3D HLSL) don't 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.

An oldest solution to combining effects, used even before the shading languages, was to perform 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 — you can only modify the complete result of the previous pass.

One 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 it doesn't allow to modify a particular calculation part without replicating the algorithm outside of this calculation. For example, if you want to scale the light contribution by a shadow function, you will have to also replicate the code iterating over the light sources.

Another common solution is to arrange shaders in a pipe, where one shader processes the result of another. This can be visualized as layers of materials, where each layer modifies the previous color.

It is actually similar to a multi-pass rendering approach, except you don't lose speed on repeating the geometry transformation work. Still it suffers from the same limitations, as you cannot change the calculation within an existing layer, without replicating the whole algorithm of this layer.

A different approach is presented by the Sh language (<http://libsh.org/>). It allows writing shaders 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). You 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 you have a 3D engine in C++, and you want to use it in your own C++ program and extend its shaders. In this paper, we want to create a solution that is absolutely separate from the programming language used to develop a browser. Invoking a compiler to generate a final GPU shader, not to mention forcing users to learn C++, is out of the question.

OGRE (<http://www.ogre3d.org/>), an open-source 3D engine written in C++, has a system for adding shader extensions (see [OGRE]). Its idea is similar to our system (enhance the built-in shaders with your 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.

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 a hook, and to use a hook. 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 you can query the list of loaded functions by name, and call function by its name.

Drupal approach is actually quite similar to our core idea of combining effects. Our effects are similar to Drupal's modules, and our „plugging points” are analogous to Drupal hooks. Our effects can define functions with special names to enhance the standard shader behavior, just like Drupal modules can define functions to act on an event from another module. We can also define new plugs, for other effects to use. Of course we also have some special problems (shading language is quite far from a scripting language, so calling the plugs must be implemented by text replacements inside the browser) and some special opportunities (we can define effects at the appropriate nodes of X3D, like textures and lights sources, as we don't want to throw all the effects in one bag).

3 Plugs: extending the base shader

The basic idea of our approach is that the base shader code defines points when a calls to user-defined functions may be inserted. We call these places *plugs*, as they act like sockets where a 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 browser 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 example is presented in the X3D classic (VRML) encoding. You can add this inside any Appearance node:

```
effects Effect {
    language "GLSL"
    parts EffectPart {
        type "FRAGMENT"
        url "data:text/plain,
            void PLUG_texture_apply(
                inout vec4 fragment_color,
                const in vec3 normal)
            {
                fragment_color.rgb *= 2.0;
            }
    }
}
```

This defines a GLSL function named `PLUG_texture_apply` within an `EffectPart` node. The call to this function will be automatically inserted at the `texture_apply` plug point in the browser internal shader. This particular plug, the `PLUG_texture_apply`, is called after the normal texture colors are applied, but before the alpha test, and is a usual place to „just modify the pixel color”. `fragment_color` is an `inout` parameter, by modifying it you modify the color that will be displayed on the screen.

We put a short reference at the end of this paper of all the plugging points available in our shaders. For each plugging point, like this

`PLUG_texture_apply`, we define a list of parameters (you have to declare them exactly the same in your code), and we define when it is called.

Many usage scenarios are possible:

1. The Effect nodes may refer to plug names defined inside the browser 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 (you just add a „magic” comment) to define plugging points in your own shader code, and this way your own effects can be customized.
3. Inside the browser 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 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 such effects easily by themselves.

Actually, there are even more possibilities. We have been talking above about the „browser 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 browser shaders. If you define the same (or compatible) plugging points inside your shader, then the browser effects will be even added to your own shader. And of course user effects are added to your shader too. This way we have made even the standard shader nodes much more flexible. Note that if you don't define any plugs inside your `ComposedShader` node, it continues to function as before — no effects, from browser or author, will be added.

3.1 Effect node

We define new `Effect` node that holds information about the source code and uniform values specific to a given effect.

```
Effect : X3DChildNode
SFString [] language ""
    # Just like ComposedShader.language.
    # This effect will be used only when
    # the base shader (browser internal
    # or selected from Appearance.shaders)
    # has the same language.
SFBool [in,out] enabled TRUE
    # Allows to 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

# And any number of uniform variables,
# just like for ComposedShader node:
fieldType [] fieldName
fieldType [in] fieldName
fieldType [out] fieldName
fieldType [in,out] fieldName
```

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

```
EffectPart : X3DNode, X3DUrlObject
SFString [] type "VERTEX"
# Like ShaderPart.type:
# allowed values are
# FRAGMENT | VERTEX | GEOMETRY.
MFString [] url []
# The source code, like ShaderPart.url.
```

The functions that enhance standard shaders behavior are recognized by names starting with `PLUG_` in the source code. You can also freely define your own utility functions. You can pass uniform variables to the shader, and you can pass varying variables between the vertex and fragment shaders, just like with standard shader nodes.

In a single `EffectPart` node, you can define many `PLUG_` functions. However, you can only plug functions into the declared shader type. For example, you cannot use the `texture_apply` plug within a `VERTEX` shader. If your effect requires some processing per-vertex and some per-fragment, you will probably use two `EffectPart` nodes, with appropriate types. While this may seem like an arbitrary limitation, this reflects how shader parts are declared in shading languages with separate namespaces for vertex and fragment parts. 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 doesn't define any `PLUG_XXX` functions. Such `EffectPart` node may be useful for defining shading language utility functions, used by the 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. It also allows for cleaner error detection (parsing errors will be detected inside the given unit).

At one point we tried the approach to not look at any special function names in a shader code, and instead define a plug name in the separate field of the `EffectPart` node. However, the browser may need to know the full declaration of a function, to make a forward or external declaration of it. One way to overcome this problem was to force repeating this declaration in another field. Another way was to split shader code into much more parts (for example, one part declares the uniform variables, one part declares the plug function, one part defines the function body and so on). Both approaches seemed uncomfortable and not natural for authors, and they didn't really offer much simpler implementation, so we dropped this idea. The current approach, to find the declarations of `PLUG_XXX` functions inside a complete shader code, is easy to implement, and results in clean shader code of the effects. It also allows us to naturally use the separate compilation units in case of GLSL.

3.2 Effects for particular shapes appearance

There are various places where an `Effect` node may be used. If it's specific to a given shape appearance, you can place it on the `Appearance.effects` list:

```
Appearance
MFNode [] effects [] # Effect
```

All the effects on this list (with suitable language) will be used. Note that this is different than the `Appearance.shaders`, which chooses only one shader. For effects, we choose all of them.

The most basic idea of this paper is to allow you to define two independent shader effects, and then seamlessly connect them by simply placing them both on the `Appearance.effects` list. This also allows to define a library of effects, that can be composited without any work needed by user.



Figure 2: Toon and Fresnel effects combined.

Note that all our nodes benefit from X3D mechanism to reuse the nodes by reference (the `DEF` / `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, you can create an `EffectPart` with a library of useful shading language functions, and reuse if for various effects.

3.3 Effects for a group of nodes

Our `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, a browser can implement the `LocalFog` node as a prototype that expands to our `Effect` node. This gives you a 100% correct implementation of the standard `LocalFog` node, as is trivially easy.

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 [“Humus” 2006]). 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. Since it is implemented as an effect, it can be instantly used with various lighting and texturing conditions — it simply works for all X3D shapes.

3.4 Light sources effects

The real power of our system comes from the fact that some nodes, like light sources and textures, can define their own effects. So you can easily define a shader specific for a given light source, or texture.

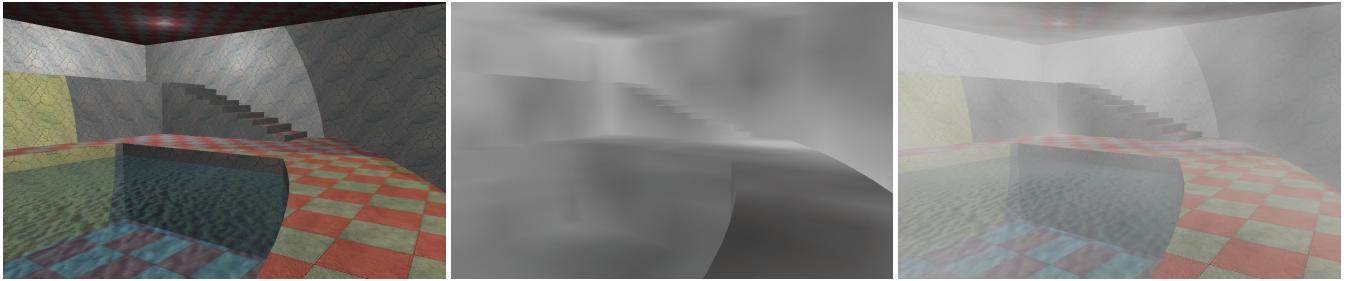


Figure 3: Volumetric fog scene: 1) No fog; 2) No lighting; 3) Lights and fog. Note that the fog is assumed to have its own ambient lighting, so it colors the image even in the 2) case.

We add the `effects` field for every light node:

```
X3DLightNode
MFNode [] effects [] # Effect
```

This allows you to modify the light source contribution of this particular light node. For example you can modify the spot light shape, possibly based on some texture information. Or you can implement a different lighting model, like anisotropic Ward or Cook-Torrance.

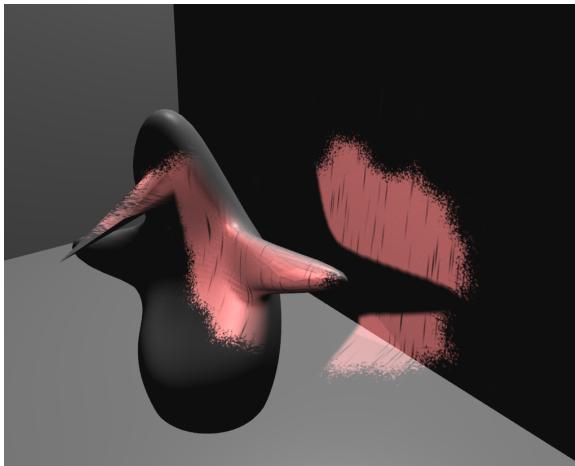


Figure 4: Textured spot light with shadow.

3.5 Texture effects

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

```
X3DTextureNode
MFNode [] effects [] # Effect
```

You can use the `effects` field inside any X3DTextureNode to enhance and modify the look of any standard texture node, like `ImageTexture`. You can use a plug `texture_color` to change the texture color, knowing the current texture coordinates and other information.

3.5.1 ShaderTexture effects

We introduce a new X3D node designed specifically for generating textures using the shaders. This is suitable if your texture is de-

fined completely using the shading language. The texture contents are not stored anywhere (not even on GPU), and the X3D browser doesn't manage any texture resources. From a GPU point of view, there is no texture¹. There is only a shader function that generates colors based on some vectors. By wrapping such function inside the new `ShaderTexture` node, you can treat it much like other X3D textures. In particular, you can provide texture coordinates (explicit or generated) for the texture. Effectively, it behaves like a normal texture node, with all the related X3D features.

```
ShaderTexture : X3DTextureNode
MFNode [] effects [] # Effect
SFString [] defaultTexCoord "BOUNDS2D"
# ["BOUNDS2D" | "BOUNDS3D"]
```

Actually, the `effects` field is already defined at the `X3DTextureNode` class (see above). We just mention it here for completeness.

You should include an effect overriding at least the `texture_color` plug, otherwise texture contents are undefined. Our implementation actually sets the default texture color to pink (RGB(1, 0, 1)), so it stands out, reminding you to override it.

The `defaultTexCoord` field is used only when a particular geometry doesn't specify the texture coordinates to use for this texture:

1. "BOUNDS2D" means that the algorithm described in the specification for `IndexedFaceSet` should be used. This adapts 2D texture coordinates to the two largest bounding box sizes. It's most suitable for 2D textures that use only 2D coordinates (ignoring other coordinates, or assuming they are always 0, 1).
2. "BOUNDS3D" means that the algorithm described in the specification of `Texturing3D` component (section "*Texture coordinate generation for primitive objects*") is used. This adapts 3D texture coordinates to the bounding box sizes. It's most suitable for 3D textures (you can ignore the 4th texture coordinate component, or treat it as homogeneous, as "BOUNDS3D" will always set it to 1).

Remember that the texture coordinates can be always explicitly specified at the geometry, by any of the `X3DTextureCoordinateNode` nodes. The `defaultTexCoord` says only what is the default approach, when you don't use `texCoord` at the geometry node. The idea is that using a `ShaderTexture` should be as comfortable as any other texture node.

¹But the spoon is real, we swear.

3.5.2 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.

Remember 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 uniform value. This allows you to base your effects calculation on any number of textures, and combine them in any way you like. The only difference between `ShaderTexture` and other textures is what the browser does automatically for you, that is what color is passed to the first `texture_color` plug.

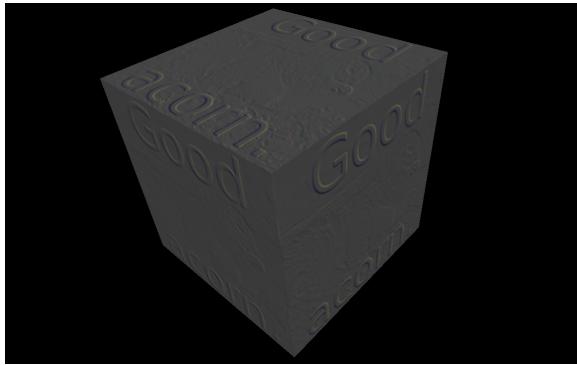


Figure 5: `ShaderTexture` doing an edge detection operation on a normal `ImageTexture`.

3.5.3 Independence from texture filtering

Note that the shader effects for textures are calculated at each screen fragment (not at each texel). So your effects are not concerned about the texture size or texture filtering options. You just use the interpolated texture coordinates in the `texture_color` plug.



Figure 6: Yellowish arc is done by a texture effect, as so is not affected by the pixelated look of the base image texture.

4 Defining your own plug points

In your shader code, you can define new plug points 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`, you can even use the same `PLUG_name` many times within a single effect. 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 appearance, and finally the effects inside the grouping nodes. Although, preferably, for most effects this order will not matter.

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 declared many times in the source shader. This means that a single `PLUG_XXX` function will be called many times. This is typically useful when your shader calculation is naturally expressed as a loop, but you had to unroll this loop for shader source (for example, to slightly tweak some loop iterations). The plug names that are available per-light source and per-texture are an example of this. If you use the `PLUG_light_scale` inside `Appearance.effects`, you change the intensity of all the light sources on the given shape. Contrast this with using the same `PLUG_light_scale` inside a `X3DLightNode.effects`, where you only change the given light node contribution.

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.

The nice feature of our magic `/* PLUG ... */` comments is that a shader source is still valid even if you completely ignore the plugs. For example, you can write a custom `ComposedShader` node, defining some plugs, and for browsers that understand them — the plugs can be used, for other browsers — plugs will be gracefully ignored (but still the shader will run, although without any effects).

You can define new plug points in your own effects code, as well as in your complete custom shaders (like `ComposedShader`) code. In the first case, the plug points are only available for the following effects of the same node.

4.1 Where the forward declarations are placed

Special comment `/* PLUG-DECLARATIONS */` may be used near the beginning of your shader source code. It is only useful if

your shader code defines any new plugging points, that is if you have any magic `/* PLUG: ... */` comments inside. When some other effect uses your plugging point, the browser adds an appropriate call to a function in that effect. Additionally, the browser has to declare the function, because it may be in a separate compilation unit (in case of GLSL), or just defined later in the code. These (forward or external) declarations are inserted at the point of special `/* PLUG-DECLARATIONS */` comment, or (when it is missing) simply at the beginning of your shader source. This applies in the same way to shader code inside an `EffectPart`, or inside standard X3D node like `ShaderPart`.

Using `/* PLUG-DECLARATIONS */` may be necessary as 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. You should place `/* PLUG-DECLARATIONS */` after such directives, and before any `/* PLUG: ... */` declarations.

4.2 Invalid shader code

We guarantee the behavior only if the provided shading language code is a correct, self-contained code. X3D browser doesn't validate this code in any way, so any error (like undeclared variable, unterminated block (,,{ without matching ,})), or an unterminated comment) may be only detected after the complete shader is determined and compiled by the GPU.

Although in case of shading languages with separate compilation units, the separation is actually better, and parsing errors will cause problems in your own code only. Still, by writing incorrect code, you can cause the whole shader to malfunction.

In all our practical tests, this approach didn't cause any problems. Since you code each effect separately, you also test them separately, and in practice it's usually obvious what problem causes the parsing error. This is also a direct consequence of our decision to **never require the browser to parse the shader code**.

It should be noted however that in particularly nasty cases, a deliberately poorly coded effect may cause troubles for other effects. In particular, since you can use `#define` and macros in your effect code, you can do nasty tricks to break other effects. You can make them compile, but function incorrectly. However, we don't consider it a real problem. You really have to deliberately want to do something bad, and be familiar with internals about how the shader is generated, to achieve some particular weird behavior. It doesn't happen by accident in our experience.

Note that this isn't a security problem — bad shader code only breaks rendering of a particular shape. And X3D `ComposedShader` node allows users to execute any shading language code anyway. So if there's anything dangerous (for example a buggy OpenGL may cause the browser process to exit with segmentation fault on some special shader code snippets), you could do it without using our effects framework anyway.

5 Various examples

Remember that effects may define their own uniform variables, just like the standard shader nodes. So you can pass your own textures to effects. For example you can write an effect that mixes a couple of textures, using any information available in the shader as a criteria for mixing. You can also pass any other values for an effect, for example you can pass the current time from a `TimeSensor` and make dynamic effects.

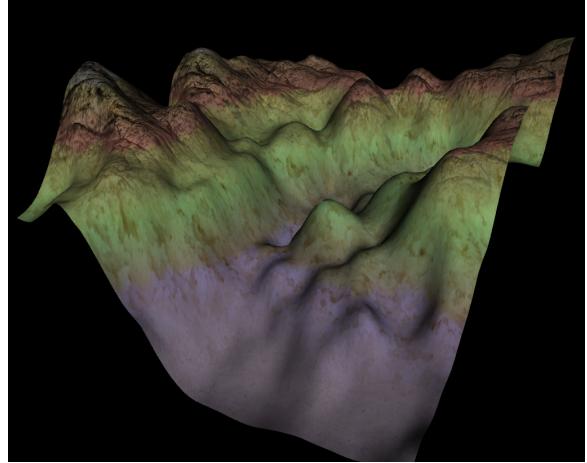


Figure 7: *ElevationGrid with 3 textures mixed inside the shader.*

We can also wrap 2D and 3D noise inside a `ShaderTexture`. A texture node like `NoiseTexture` from InstantReality may be implemented on GPU by a simple prototype using the `ShaderTexture`.

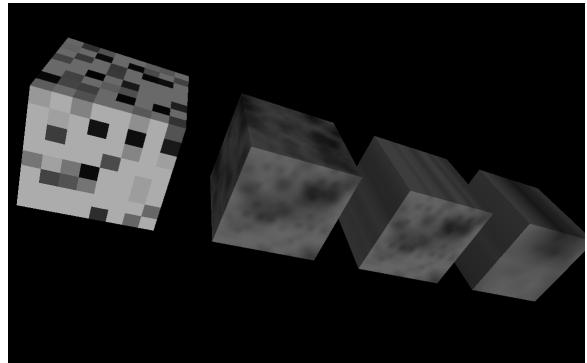


Figure 8: *3D and 2D smooth noise on GPU, wrapped in a ShaderTexture.*

Another example is *water*. Water is very nice to implement by our effects, as a proper water simulation is naturally a combination of a couple effects. You want to simulate waves, so you want to vary vertex heights, or vary per-fragment normal vectors (for best results, you want to do both things). You 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 effects.

Our approach also allowed us to easily implement and test two alternative versions for generating water normals. One approach was to take normals from the pre-recorded sequence of images (encoded inside `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. Thanks to our effects system, we could immediately test our alternative normal vector approaches, without touching the water reflection / refraction effect. Moreover, the implementations of these two approaches are concerned only with calculating the normal vectors in the object space. They override a special plug of yet another effect, that transforms these normal vectors into the eye space. This way we have extracted all the common logic into a separate effect, making



Figure 9: Water using our effects: 1) Per-pixel lighting and bump mapping. 2) Per-pixel lighting and reflections and refractions (by a single environment cube map texture). 3) All effects.

it clear where the alternative versions differ and what do they have in common.

We also provide plugs to easily change the geometry in object space. Since the transformation is done on GPU, there's practically no speed penalty for animating thousands of flowers in our test scene.



Figure 10: 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 by the standard X3D *Programmable shaders component*. However, such implementation would be extremely cumbersome. You would first have to prepare a shader code to calculate all the (multi-)texturing, lighting, shadows, and all the other effects that you use in your scene. This is a large work if you consider all the X3D rendering options, and note that a shader should remain optimized for a particular setting. Actually, it isn't even possible, unless you can calculate some global options, like which light sources and fog nodes affect the given shape. 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. And all the information is nicely integrated with X3D nodes, effects are specified at suitable nodes, and their uniform values and attributes are integrated with X3D fields.

6 Implementation notes: GLSL

We have implemented the effects described here only for the *OpenGL Shading Language* (GLSL). However, we have designed our extensions to be applicable to other shading languages as well

(like Cg or HLSL), and we believe they can be handled in a similar fashion.

One important feature that makes GLSL slightly different from other shading languages is the *separate compilation units*. It's a useful feature, as it forces cleaner shader code (you cannot use undeclared functions from other shader parts) and it gives you better line numbers in error messages (although a pre-processor directive like `#line` could also be used for this). But it's not necessary for a working implementation of our effects — we know it because our early implementation was keeping the full generated shader code inside one unit, and this approach was also working correctly.

7 Short 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. See our engine website, in particular the online reference of our plugs on http://vrmlengine.sourceforge.net/compositing_shaders.php, for the most up-to-date list.

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.

7.1 Vertex shaders 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 in 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.

7.2 Fragment shaders 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.

```
void PLUG_light_scale(
    inout float light_scale,
    const in vec3 normal_eye,
    const in vec3 light_dir,
    const in gl_LightSourceParameters light,
    const in gl_LightProducts light_products,
    const in gl_MaterialParameters material)
```

Scale the given light source contribution. This plug is also available at light source nodes' effects.

```
void PLUG_add_light_contribution_front (
    inout vec4 color,
    const in vec4 vertex_eye,
    const in vec3 normal_eye,
    const in gl_MaterialParameters material)
```

Add pixel color for a lit material. This is typically used to add the light sources. There is also the `add_light_contribution_back`, for light contribution on the back side of the faces.

```
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_texture_apply(
    inout vec4 fragment_color,
    const in vec3 normal_eye)
```

At this point, the textures are applied. You can change the fragment color now, for various effects.

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

At this point, the X3D fog is applied. Again you can change here the fragment color, as you desire. This plug is called after the `texture_apply`, because you usually want to apply the fog to the final (textured) fragment color.

```
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.

8 Complete example models

Examples models using our effects are available inside our engine demo models on http://vrmlengine.sourceforge.net/demo_models.php. The relevant demos are mostly inside the `compositing_shaders` subdirectory, also the `water` subdirectory contains the water implementation using our effects. You can open the examples using any of our engine tools, like `view3dscene`.

You can run `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. 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.

Acknowledgements

A lot of people helped and encouraged the development of our VRML/X3D engine, with its rendering features and extensions. A big „thank you” goes to all of you.

The Japanese shrine model, used for an eye-candy screenshots at the beginning of this paper, comes from <http://opengameart.org/content/shrine-shinto-japan>.

References

“HUMUS”, E. P., 2006. Volumetric Fogging 2. <http://www.humus.name/index.php?page=3D&ID=70>; nice overview also on <http://www.evl.uic.edu/sjames/cs525/shader.html>.

OGRE. OGRE Wiki - RT Shader System. <http://www.ogre3d.org/tikiwiki/RT+Shader+System&structure=Development>.

WEB 3D CONSORTIUM, 2008. Extensible 3D (X3D) Graphics Standard. ISO/IEC 19775-1.2:2008; see <http://web3d.org/x3d/specifications/>.