EG UK Theory and Practice of Computer Graphics (2011)
Hamish Carr, Ian Grimstead (Editors)

Compositing Shaders in X3D

M. Kamburelis[1]

[1]Institute of Computer Science, University of Wrocław, Poland

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. 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, light sources and textures.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; Computer Graphics [I.3.6]: Methodology and Techniques—Languages, Standards

1. Introduction

X3D [Web08] is an open standard for representing interactive 3D models, with many advanced graphic features.

The X3D *Programmable shaders component* [dCGP04] (part of the X3D standard) defines how *shaders* can be assigned to particular 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.

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 implementation usually 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. The *light_color* function may be replaced to use different lighting models. 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.

Figure 1: *Japanese shrine model with more and more effects applied: Phong shading (per-pixel lighting), bump mapping, shadows from two lights. The model is based on* `http://opengameart.org/content/shrine-shinto-japan`.

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.

2. Motivation and previous work

The most popular real-time shading languages right now are OpenGL `GLSL` [Ros04], NVidia `Cg` and Direct 3D `HLSL`. They 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* [dCGP04] 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 solution 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. 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/`, [TM04]) allows writing shader code (that can run on GPU) directly inside a C++ program. 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 [OGR]). 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 [KRSH10] allows to integrate internal renderer shaders with user shaders, by introducing a new shader language.

Spark [FH11] is a recent work presenting a new language to develop composable shaders for GPU.

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. 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.

3. Extending the shaders with plugs

The core idea of our approach is that the base shader code 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 `view3dscene`.

```
#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_texture_apply(
        inout vec4 fragment_color,
        const in vec3 normal)
      {
        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_texture_apply` indicates that we use the `texture_apply` plug. This particular plug is called right after applying the textures, and is the simplest way to "just modify the 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 on `http://vrmlengine.sourceforge.net/compositing_shaders.php`. For each plug, like this `PLUG_texture_apply`, 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 [Kam10]) and the bump mapping to use our "plugs" approach. This made their implementation very clean, short and nicely separated from each other.

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.

3.1. 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 [Web08].

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.
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 [dCGP04].

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.
```

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 `texture_apply` 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).

3.2. Effects for particular shapes

There are various places where an `Effect` node may be used. To apply an effect for a given shape appearance, 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 2: *Toon and Fresnel effects combined.*

Note that all introduced 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, the same `EffectPart` node, with a library of useful shading language functions, may be used for many effects.

3.3. 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 [Per06]). 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 3.

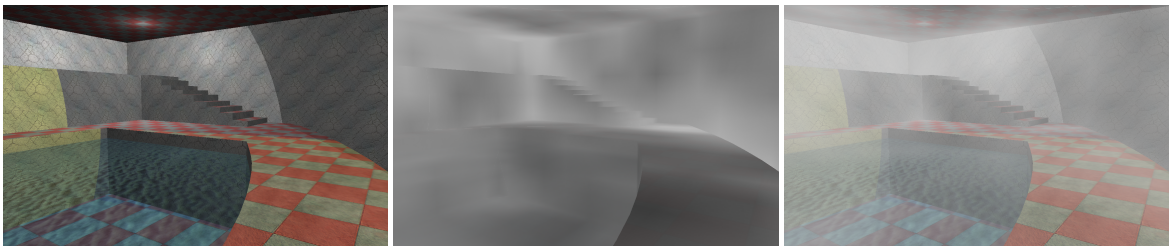

Figure 3: *Volumetric fog scene: 1) No fog; 2) No lighting; 3) Lights and fog. Note that the fog is assumed to have its own ambient lighting, so it colors the image even in the 2) case.*

3.4. 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 4). 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
```

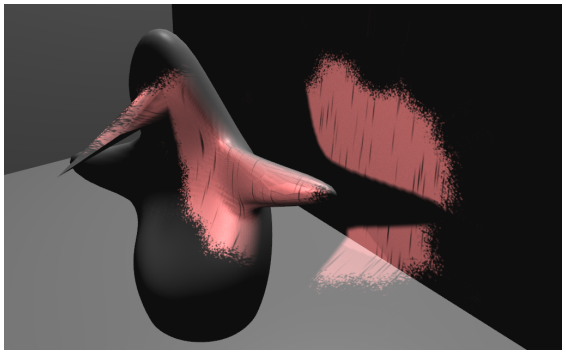

Figure 4: *Textured spot light with shadow.*

3.5. Texture effects

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

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

`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.

3.5.1. 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.

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

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. When the texture coordinates are not explicitly given, the `defaultTexCoord` field determines how they are generated. `"BOUNDS2D"` generates 2D texture coordinates, adapting to the two largest bounding box sizes (the 3rd texture coordinate is always 0). This is most comfortable when the texture color depends only on the XY components of the texture coordinate. The precise behavior of `"BOUNDS2D"` follows the X3D `IndexedFaceSet` specification and the precise behavior of `"BOUNDS3D"` is described in the *Texturing3D* component of the X3D specification.

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.

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.

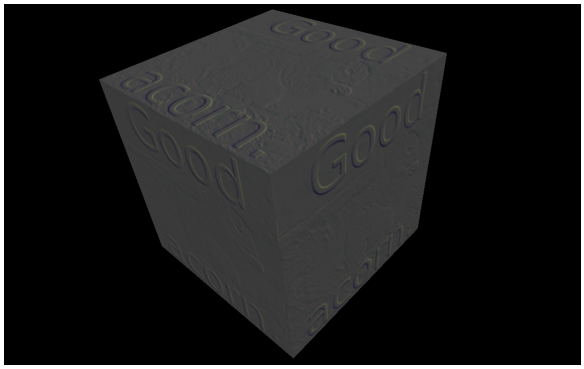

Figure 5: *ShaderTexture doing an edge detection operation on a normal ImageTexture.*

4. 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.

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. This means that a single `PLUG_xxx` function will be called many times. For example, 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). The plug names that are available per-light source and per-texture are an example of this. Using the `PLUG_light_scale` inside `Appearance.effects`, the intensity of all the light sources on the given shape can be changed. Contrast this with using the same `PLUG_light_scale` inside a `X3DLightNode.effects`, where only the given light node contribution is changed.

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

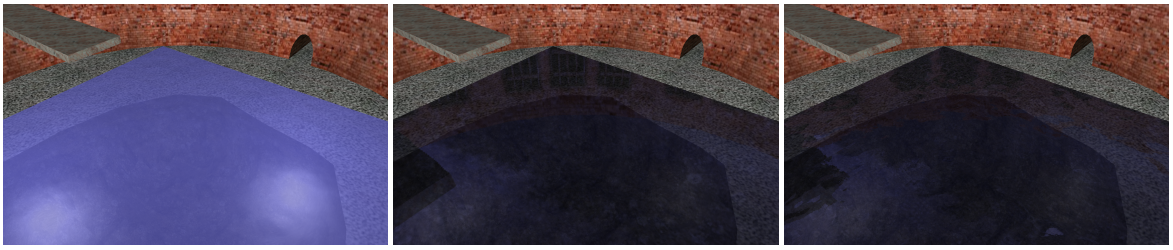

Figure 8: *Water using our effects: 1) Per-pixel lighting and bump mapping. 2) Per-pixel lighting and reflections and refractions (by a single environment cube map texture). 3) All effects.*

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.

4.1. Forward declarations

Suppose we have an effect X that defines a new plug. 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.

4.2. 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).

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.

5. Examples

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 6). 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.

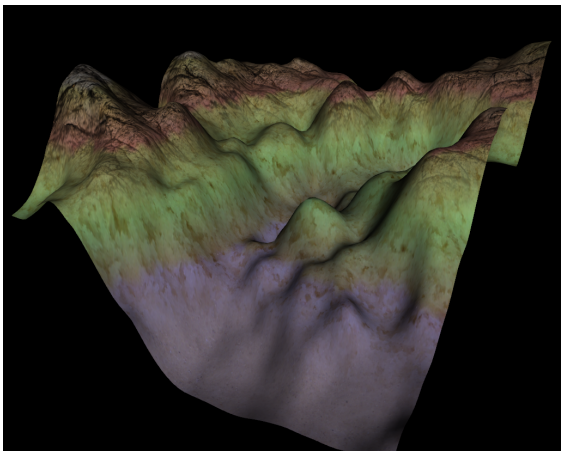

Figure 6: *ElevationGrid with 3 textures mixed inside the shader.*

We can wrap 2D or 3D noise inside a `ShaderTexture` (see figure 7). A texture node like `NoiseTexture` from InstantReality [Ins] may be implemented on GPU by a simple prototype using the `ShaderTexture`.

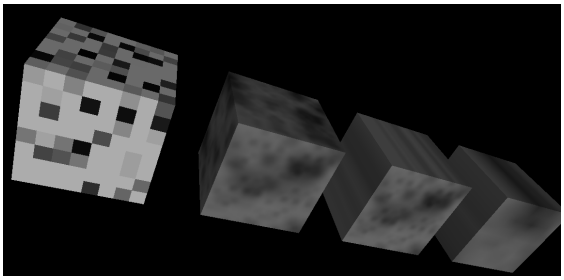

Figure 7: *3D and 2D smooth noise on GPU, wrapped in a ShaderTexture.*

Water is very nice to implement with the help of 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. Then we have tested two alternative approaches for normal generation (take from pre-recorded series of normal-maps, or calculate from a smooth 3D noise). They both generate normal vectors in the tangent space, overriding a plug defined by yet another effect that transforms normals into the eye space. This way we have extracted all the common logic into a separate effect, making it clear where the alternative normal generation methods differ and what they have in common.

We also have plugs to 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. See figure 9.

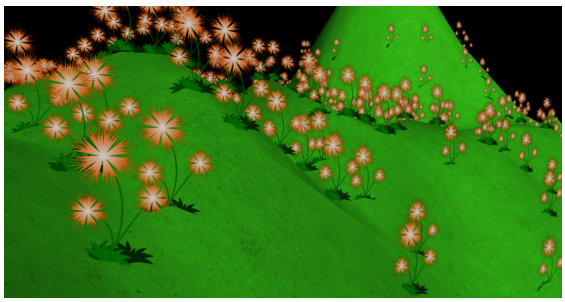

Figure 9: *Flowers bending under the wind, transformed on GPU in object space.*

We would like to emphasize that all the effects demonstrated here are theoretically already possible to implement using the standard X3D *Programmable shaders component* [dCGP04]. 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 nodes, effects are specified at suitable nodes, and their uniform values and attributes are integrated with X3D fields.

Many complete example models using our effects are referenced from our page on `http://vrmlengine.sourceforge.net/compositing_shaders.php`. You can open these examples using any of our engine tools, like `view3dscene`.

6. Implementation notes

We have implemented all the X3D extensions described in this paper 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. In particular, we have tested that the *separate compilation units* concept of GLSL, while very useful, is not necessary for proper implementation of our effects.

7. 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.

References

[dCGP04] DE CARVALHO G. N. M., GILL T., PARISI T.: X3D programmable shaders. In *Proceedings of the ninth international conference on 3D Web technology* (New York, NY, USA, 2004), Web3D '04, ACM, pp. 99–108. 1, 2, 4, 8

[FH11] FOLEY T., HANRAHAN P.: Spark: Modular, Composable Shaders for Graphics Hardware. In *Proceedings of SIGGRAPH 2011* (2011), ACM. 2

[Ins] INSTANT REALITY: NoiseTexture. `http://doc.instantreality.org/documentation/nodetype/NoiseTexture/`. 7

[Kam10] KAMBURELIS M.: Shadow maps and projective texturing in X3D. In *Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), Web3D '10, ACM, pp. 17–26. 3

[KRSH10] KARRENBERG R., RUBINSTEIN D., SLUSALLEK P., HACK S.: AnySL: efficient and portable shading for ray tracing. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG '10, Eurographics Association, pp. 97–105. 2

[OGR] OGRE: OGRE Wiki - RT Shader System. `http://www.ogre3d.org/tikiwiki/RT+Shader+System&structure=Development`. 2

[Per06] PERSSON E.: Volumetric Fogging 2. `http://www.humus.name/index.php?page=3D&ID=70`, 2006. 4

[Ros04] ROST R. J.: *OpenGL Shading Language*. Addison-Wesley, 2004. 2

[TM04] TOIT S. D., MCCOOL M.: *Metaprogramming GPUs with Sh*. A K Peters/CRC Press, 2004. 2

[Web08] WEB3D CONSORTIUM: Extensible 3D (X3D) Graphics Standard. ISO/IEC 19775-1.2:2008; see `http://web3d.org/x3d/specifications/`, 2008. 1, 3