Compositing Shaders in X3D

Michalis Kamburelis
michalis.kambi@gmail.com

Institute of Computer Science
University of Wrocław, Poland

TPCG 2011

Uniwersytet Wrocławski

Outline

Outline

Introduction

How to create and composite powerful graphic effects using shaders.

- Extend two existing languages (X3D and GLSL).
- Practical, implemented. 100% finished open-source implementation.

Easy and works

Easy and new.

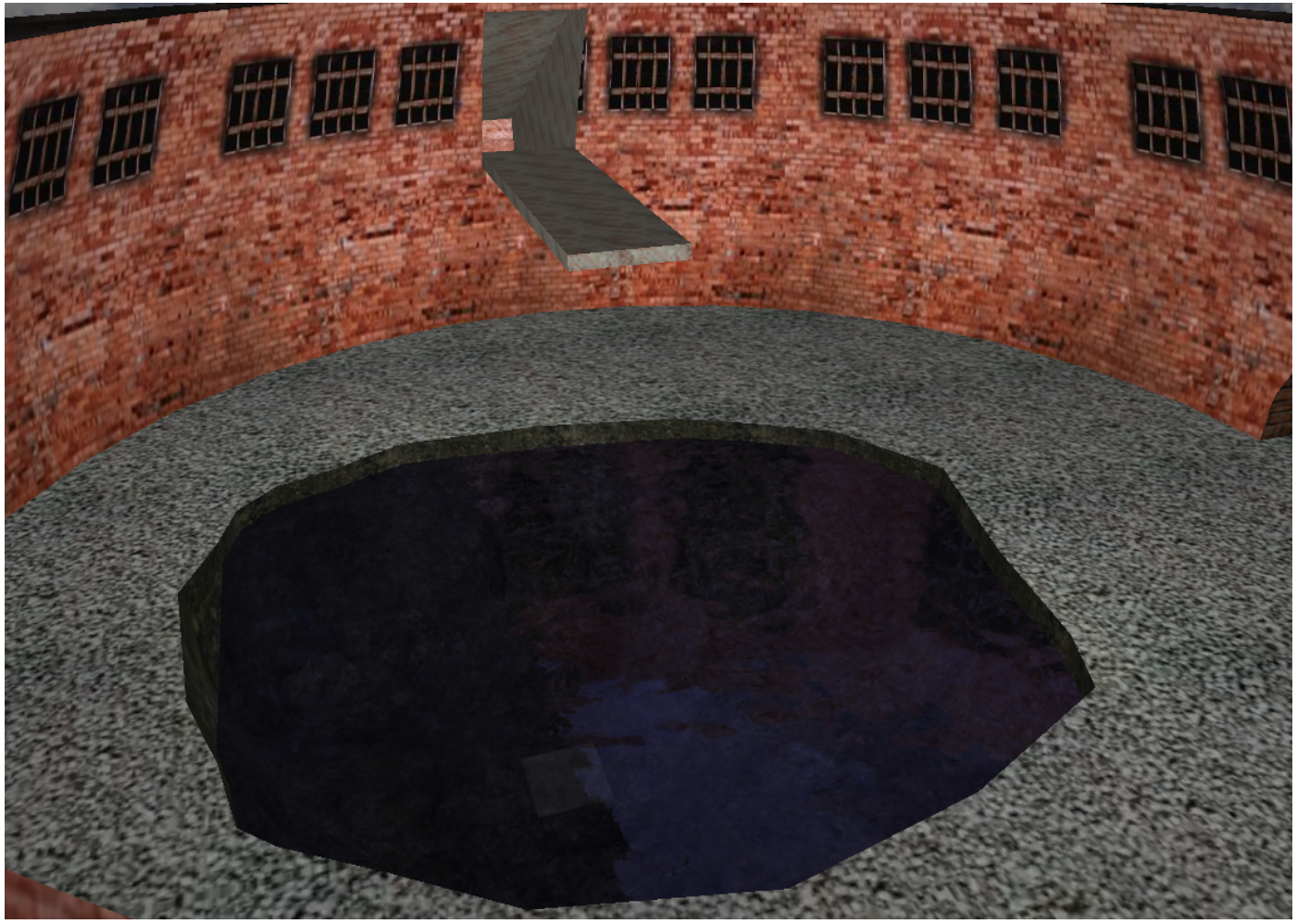
- Actually there are other approaches that do this by inventing new languages (Sh, Spark). But using them means that your renderer is designed, from the start, for the (only one) implementation of this language (like libsh). As you don't want to implement new language yourself in an already complicated renderer.
- We propose much simpler approach, no new language, just a natural extension for X3D and GLSL. And you probably already use GLSL together OpenGL.

Works perfectly :) Various shading algorithms become easy to implement, and the implementation is automatically reusable. No need to reimplement your shaders for each scene/configuration.

Example - water

Water is naturally a combination of a couple of effects: waves smoothly change normal vectors and vertex positions, also water is a little transparent and a little reflective.

Example - water - wider view



Outline

X3D

- **Extensible 3D**: language to describe 3D worlds.
- Open (full specifications on `http://www.web3d.org/`), popular standard.
- Easy to express all typical 3D world features.
- In simple cases, a tree of nodes. In a general case, a directed graph of nodes, cycles possible.
- Each node has a set of fields, some fields contain in turn children nodes.

Example

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

X3D - XML

Alternative example:

Example encoded in XML

```
...
<X3D version="3.2" profile="Interchange" ...>
  <Scene>
    <Shape>
      <Sphere radius="2" />
    </Shape>
  </Scene>
</X3D>
```

X3D has a lot features

That's why we chose to integrate our concept with X3D. Besides standard rendering features (indexed geometries with materials and animations), the X3D specification includes many advanced 3D visualization stuff. For example:

- NURBS,
- advanced texturing (cube maps, 2D and 3D textures, multi-texturing),
- particle effects,
- physics,
- skinned humanoid animation.

X3D as a programming language

There are some aspects of X3D that make it more interesting, more like “(declarative) programming language” than just “3D model format”:

- DEF/USE: references, for all nodes.
- Events: sending and receiving events. E.g. open a door when user presses a handled.
- Prototypes: define new nodes, by combining existing ones.
- Scripts: integration with script languages (like JavaScript) easy.

Shaders

- Languages to calculate shading color of 3D stuff. Implement per-vertex work, let GPU rasterize, implement per-pixel work.
- For real-time (implemented on GPU) shaders, there are three popular choices: GLSL (OpenGL), Cg (NVidia: OpenGL or Direct 3D), HLSL (Direct 3D).
- Of course, you can use shaders in X3D.

Example X3D + GLSL

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

Shader replaces the calculations

- \+ Easy implementation for renderer, since this is how GPU works, this is what OpenGL exposes etc.
- \- **Difficult implementation of your own shaders**. Before implementing your effects, first learn to recreate the necessary features from fixed-function. Sure, you can let some renderer auto-generate a large shader for you, and only modify it, but...
- \- **You create one-time shaders, not reusable**. Particular shader is tied to a scene configuration, like the number and type of lights and textures.
- \- **All the effects in one huge shader**. Ultimately, this means that you create a large shader with `#ifdefs`, that is not maintainable. Adding / removing an effect means carefully inserting logic into a large code.

Outline

Why

- We would like to intensively use shaders. Every 3D property should be programmable — that's the reason why shaders were created.
- Traditional approach makes it difficult to actually harness that power. Implementing a simplest change (like "filter light through a stencil") requires understanding and taking care of the algorithm around.
- We prefer to create effects that extend/modify existing behaviors.

Solution

Idea 1

Easy to use mechanism to define and use “sockets” (plugs) — places where you can insert your own calculations. To use a socket named `texture_apply`, just define (in GLSL) function with a magic name `PLUG_texture_apply` in a new node `Effect`.

- We keep the full power of the shading language (GLSL).
- Implementation in renderer easy: we extend existing language.
- Implementation of new effects easy: implement your desired effect, and specify how it is tied to the existing calculations.
- Effects are reusable: inserted into an internal shader, optimized for particular textures, lights etc.
- Effects can be composited: multiple effects may refer to the same sockets.

Example of our effect

```
# inside an Appearance from previous example:
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;
    }"
  }
}
```

Details

- New node `Effect`: choose shading language, like GLSL.
- New node `EffectPart`: choose type, like vertex, pixel (fragment), etc.
- You can define new uniform values in an `Effect` node, for example pass a texture or current time (`TimeSensor.time` in X3D) to the effect.
- Effects are added to the base internal shader.

Example - compositing two trivial effects

Figure: Composite Fresnel effect and toon shading

Code: `demo_models/compositing_shaders/fresnel_and_toon.x3dv`

Internal effects

Renderer can implement internal effects in a similar way. This allows for really nice orthogonal implementation of e.g.:

- Shadow maps: filter particular light source,
- Classic bump mapping: change normal based on a texture,
- Fog: mix final color with a fog color.

Standard X3D `ComposedShader` node modifies the base shader, so it's also better now.

Figure: Compositing two shadow maps and bump mapping

Light sources effects

Idea 2

Not only visible 3D shapes have effects. Light sources also have their own effects, that may change light shape, equation etc. Such light becomes a fully-fledged, reusable X3D light.

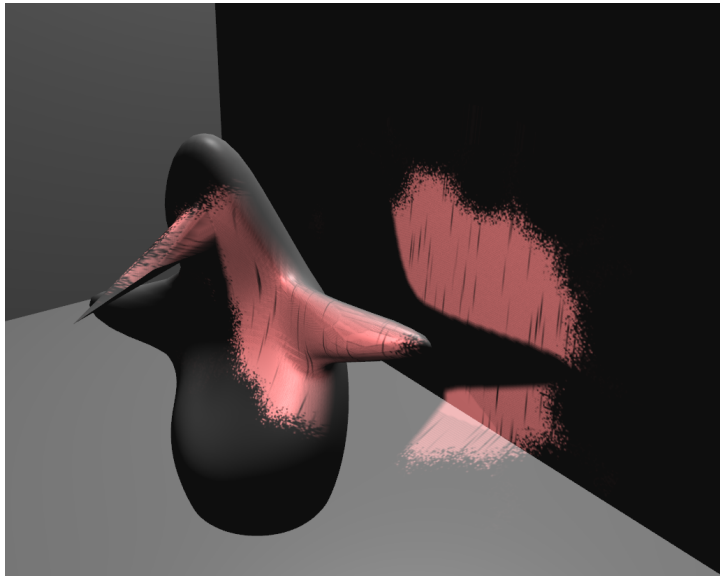


Figure: Light filtered through a texture. Automatically casts a proper shadow, and cooperates with bump mapping.

Texture effects

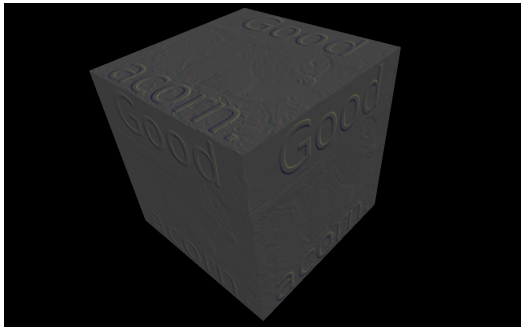
Textures can naturally also be modified using the effects. A lot of possibilities, as textures are just 2D and 3D containers for general data in many algorithms.

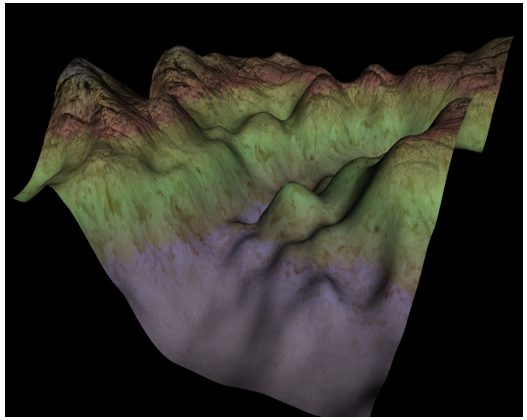
E.g. add noise to an existing texture, or choose a 2D slice of 3D texture.

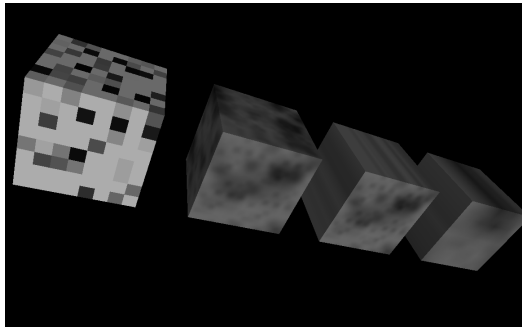
Procedural textures on GPU

Procedural texture is just a function 2D or 3D $\rightarrow$ color, normal vector, and such. There is no related storage on GPU, only a function in a shading language.

In standard X3D there is no way to treat a procedural texture like a regular texture. This means that it's uncomfortable to automatically generate tex coords for it or mix with other textures. We introduce new `ShaderTexture` node, that nicely encapsulates a procedural texture and makes it behave like a normal texture for the rest of the scene graph.







Group effects

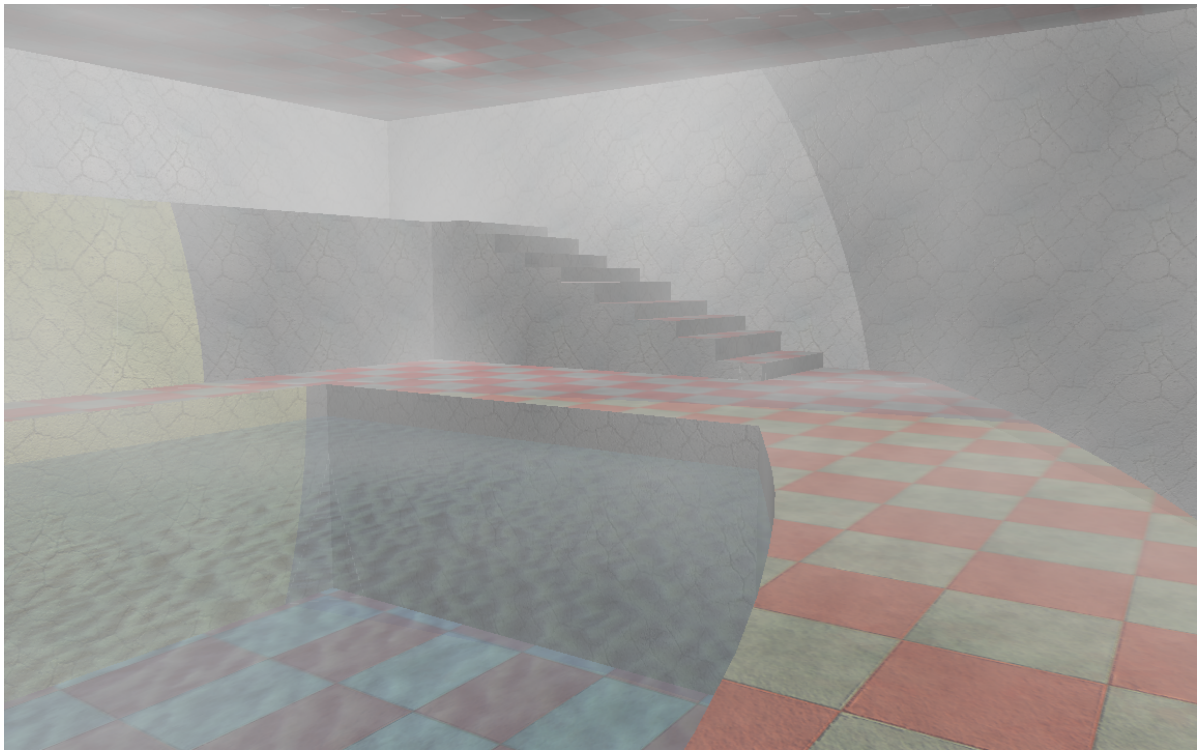


Figure: Dense animated fog (sauna).

Code: `demo_models/compositing_shaders/volumetric_animated_fog.x3dv`

Short and works on any 3D model!

Defining custom plugs

You can easily define your own plugs (sockets):

- Base shader, and **every effect**, can define plugs to be available for following effects.
  Just add a magic comment `/* PLUG: ... */` inside.
- Magic comment may be used multiple times. This enables loop unrolling in shaders.
- Magic comment `/* PLUG-DECLARATION */`. Useful to declare version and such in GLSL.
- We nicely use the *separate compilation units* of the GLSL.
- Idea portable to other shading languages.
- Good choice of default plugs, see links at the end. Despite our fear, it turns our that we don't need 100 plugs and 1 more. A couple of plugs fill the needs of 90% of use cases.

Final examples - flowers

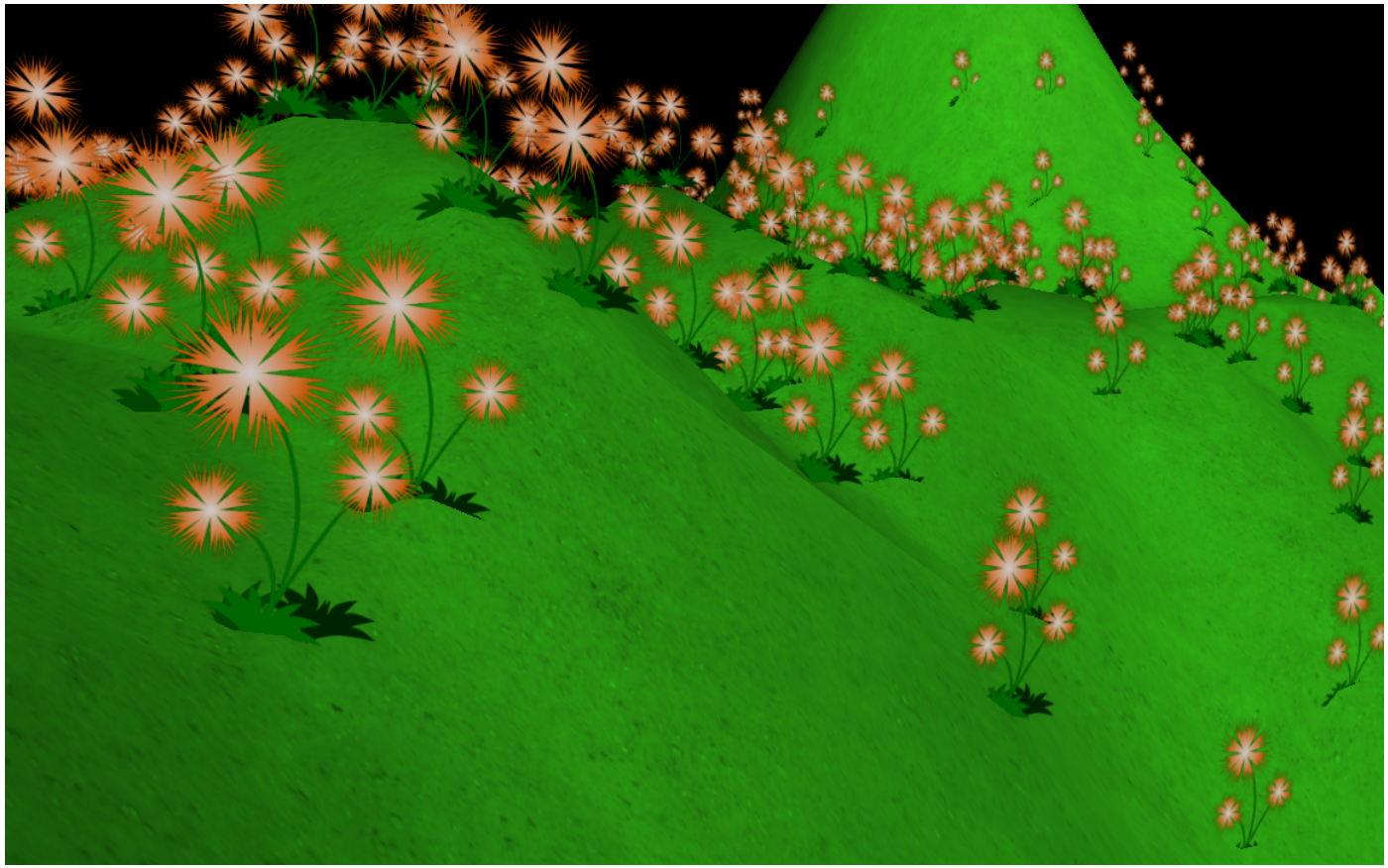


Figure: Animated flowers: transforming in object space by shaders, with zero speed cost.

Outline

Everything is implemented, open-source (LGPL), in our engine:
http://vrmlengine.sourceforge.net/
Instructions how to download and view examples are here:
http://vrmlengine.sourceforge.net/
compositing_shaders.php

Thank you for your attention!

Questions?