Komponowanie shaderów w X3D

Michalis Kamburelis
michalis.kambi@gmail.com

31 maja 2011

Uniwersytet Wrocławski

Plan

Plan

Wstęp

Jak tworzyć i łączyć efekty graficzne oparte na shaderach.

- Rozszerzenie dwóch zintegrowanych ze sobą języków (X3D i GLSL).
- Praktyczne, implementowalne (100% skończona i przetestowana implementacja open-source).

Proste i działa

Proste, ale nikt inny tego jeszcze nie zrobił :)

- Przynajmniej nie bez wymyślania zupełnie nowego języka pisania shaderów (Spark, Sh). Ale nowe języki nie zdobywają popularności, bo zazwyczaj wymagają przepisania całego renderowania. Nasz pomysł to eleganckie rozszerzenie GLSL i X3D.

Działa świetnie :) Istotnie różne algorytmy na shaderach stają się łatwe, i to do „prawdziwej” implementacji (która, raz zrobiona, musi być użyteczna na różnych modelach). Będą ładne obrazki.

Plan

X3D

- **Extensible 3D**: język do opisu światów 3D.
- Otwarty (pełne specyfikacje na `http://www.web3d.org/`), popularny standard.
- Łatwo wyrazić wszystkie typowe elementy świata 3D.
- Drzewo węzłów w wielu prostych przypadkach. Ale ogólnie skierowany graf węzłów, z możliwymi cyklami.
- Każdy węzeł ma pola, niektóre pola mogą zawierać węzły-dzieci.

Przykład

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

X3D - XML

Alternatywna wersja:

Przykład kodowania XML

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

X3D jako język programowania

Kilka elementów które czynią X3D bardziej interesującym, jako „deklaratywny język programowania”:

- DEF/USE: referencje, dla wszystkich węzłów.
- Mechanizm zdarzeń: wysyłanie i reagowanie na zdarzenia. Deklaratywny odpowiednik wywołania metody obiektu. Np. otwórz drzwi kiedy user kliknie na klamkę.
- Prototypy: można definiować nowe, pełnowartościowe, węzły jako kombinację istniejących.
- Skrypty: integracja z innymi językami (np. JavaScript) łatwa. Skrypt pozostaje „black boxem” dla X3D, jednocześnie w samym X3D definiujemy precyzyjnie do czego skrypt ma a do czego nie ma dostępu.

Shadery

Shadery:

- Języki do cieniowania obiektów 3D. Zaprogramuj pracę per-vertex, pozwól na rasteryzację, zaprogramuj pracę per-pixel.
- Nas interesują shadery na GPU, czyli do renderowania w czasie rzeczywistym. GLSL (OpenGL), Cg (NVidia: dla OpenGL lub Direct 3D), HLSL (Direct 3D).
- Naturalnie, X3D posiada węzły do definiowania shaderów.

Przykład 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 zastępuje domyślne obliczenia

- + Łatwa implementacja dla renderera, bo właśnie tak działają GPU, na to pozwala OpenGL etc.
- - **Trudne implementowanie własnych shaderów**. Zanim zaczniesz pisać swój efekt, najpierw odtwórz algorytm standardowego renderowania. Możesz skopiować kod wygenerowany przez renderer, i zorientować się gdzie dodać swoje modyfikacje, ale...
- - **Powstają shadery 1-razowego użytku**. Konkretny kod shadera przywiązuje Cię do wielu ustawień sceny (ile, jakie tekstury, światła etc.). Ogólny shader (uwzględniający wszystkie możliwości) nie jest możliwy, chociażby dlatego że nie ma będzie działał szybko.
- - **Wszystkie efekty w jednym worku**. Usuwanie / dodawanie efektu oznacza ostrożne wstawianie logiki do istniejącego (dużego) kodu.

Plan

Dlaczego

- Chcielibyśmy intensywnie używać shaderów. Każda właściwość 3D powinna być programowalna — po to stworzono shadery.
- Ale tradycyjne podejście, „napisz shader który robi wszystko”, uniemożliwia to. Zaprogramowanie najmniejszej zmiany (np. przefiltruj światło przez szablon) wymaga zatroszczenia się o algorytm naokoło.
- Wolimy programować efekty które rozszerzają/modyfikują istniejące zachowanie.

Rozwiązanie

Pomysł 1

Prosty w użyciu mechanizm do definiowania i używania „gniazd” (plugs) — miejsc gdzie można dodać własne obliczenia. Żeby użyć gniazda o nazwie `texture_apply`, zdefiniuj funkcję o magicznej nazwie `PLUG_texture_apply` w nowym węźle `Effect`.

- Zachowujemy pełną siłę języka shaderów (GLSL).
- Implementacja w rendererze łatwa: rozszerzamy istniejący język.
- Implementacja efektów łatwa: zaczynamy od razu pisać nasz algorytm, wskazujemy tylko gdzie wstawić odpowiednie elementy.
- Efekty reusable: wstawiane w wewnętrzny shader, wygenerowany znając tekstury, oświetlenie etc.
- Efekty można łączyć: wiele efektów może używać tych samych gniazd.

Przykład naszego efektu

```
# w środku Appearance z poprzedniego przykładu:
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;
    }"
  }
}
```

Szczegóły

- Nowy węzeł `Effect`: wybierz język shaderów.
- Nowy węzeł `EffectPart`: wybierz typ (vertex, pixel, etc.).
- Można definiować własne uniform w `Effect`, np. przekazać teksturę albo czas (`TimeSensor.time` z X3D) do efektu.
- Wszystkie efekty będą dodane do bazowego shadera. (Upraszczając.)
- Stary `ComposedShader` modyfikuje bazowy shader. W ten sposób stary `ComposedShader` też jest lepszy: może współpracować z wewnętrznymi efektami (shadow maps), i efektami użytkownika (z węzłów `Effect`).

Przykład - łączenie prostych efektów

Obraz: Połączenie dwóch prostych efektów Fresnela i cieniowania kreskówkowego

Kod: `demo_models/compositing_shaders/fresnel_and_toon.x3dv`

Wewnętrzne efekty

Możemy zaimplementować wewnętrzne efekty w podobny sposób.

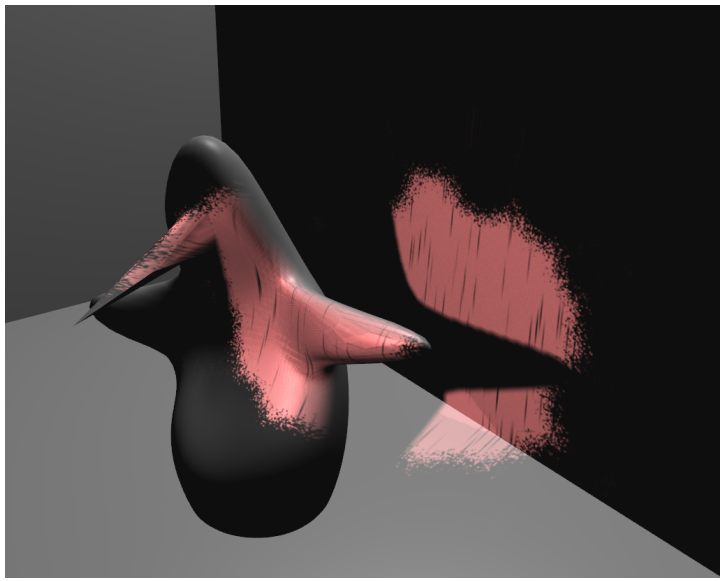
- Shadow maps: filtruj odpowiednie źródło światła,
- Klasyczny Bump mapping: zmień wektor normalny na podstawie tekstury,
- Mgła: wymieszaj końcowy kolor pixela z kolorem mgły.

Obraz: Połączenie dwóch shadow maps i bump mapping

Efekty dla źródeł światła

Pomysł 2

Efekty można dodawać nie tylko do widocznych obiektów 3D. Można zdefiniować światło które ma specyficzny kształt, równanie itp., i używać go wielokrotnie jako pełnowartościowe światło w X3D.



Obraz: Światło filtrowane przez teksturę. Rzuca poprawny cień, współpracuje z bump mapping (czego nie mamy przy prostym rzutowaniu tekstury na ścianę).

Efekty tekstur

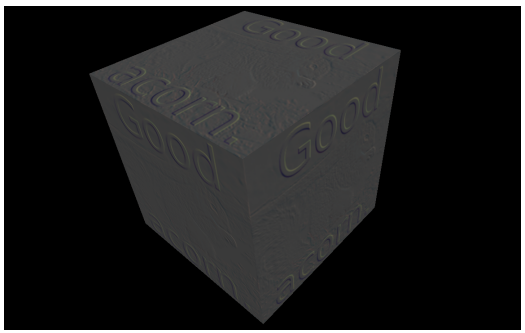
Tekstury można modyfikować przy użyciu efektów. Tekstury mają mnóstwo zastosowań (to po prostu macierz, zazwyczaj 2D lub 3D), więc wiele nowych możliwości.

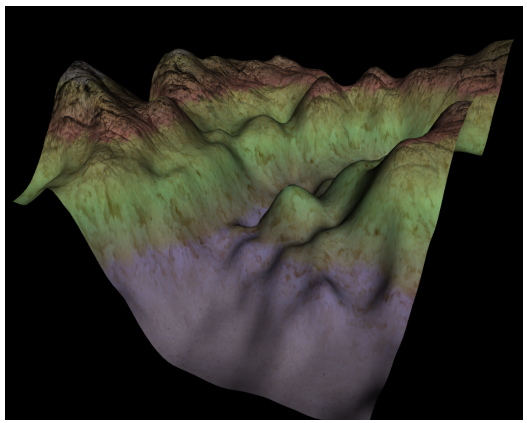
Np. zaburz istniejącą teksturę losowym szumem, albo wybierz slice tekstury 3D.

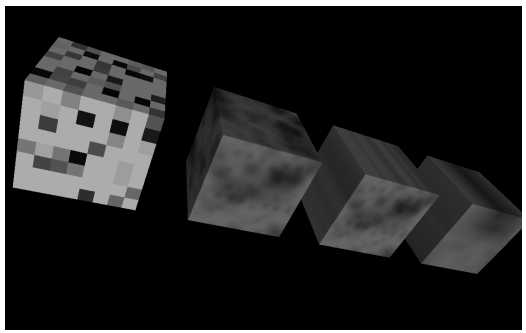
Proceduralne tekstury na GPU

Proceduralna tekstura to po prostu funkcja 2D lub 3D → kolor, wektor normalny, i inne.

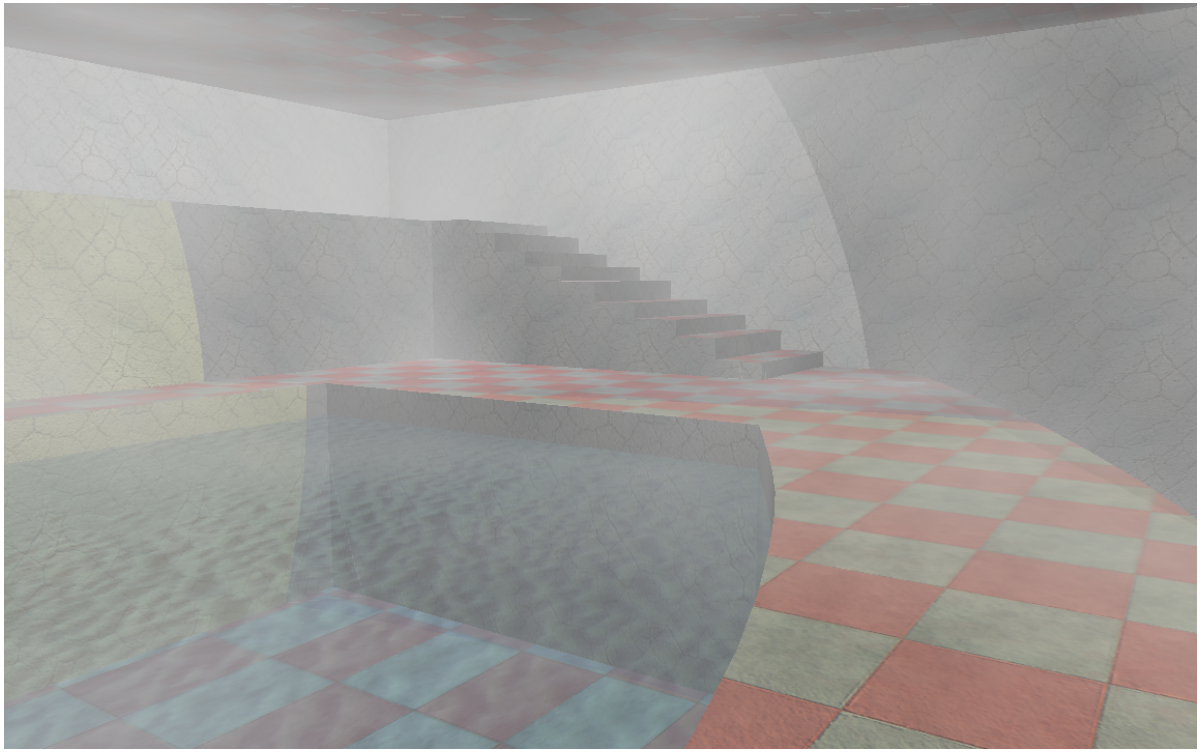
Tworzenie ich zawsze było łatwe, ale tradycyjne metody miały wadę: ponieważ nie jest to tekstura dla renderera, znikały możliwości wygodnego generowania i przypisywania współrzędnych tekstury. Rozwiązujemy to przez `ShaderTexture`.







Efekty działające na grupę



Obraz: Kłębiasta, ruchoma mgła (sauna).

Kod: `demo_models/compositing_shaders/volumetric_animated_fog.x3dv`

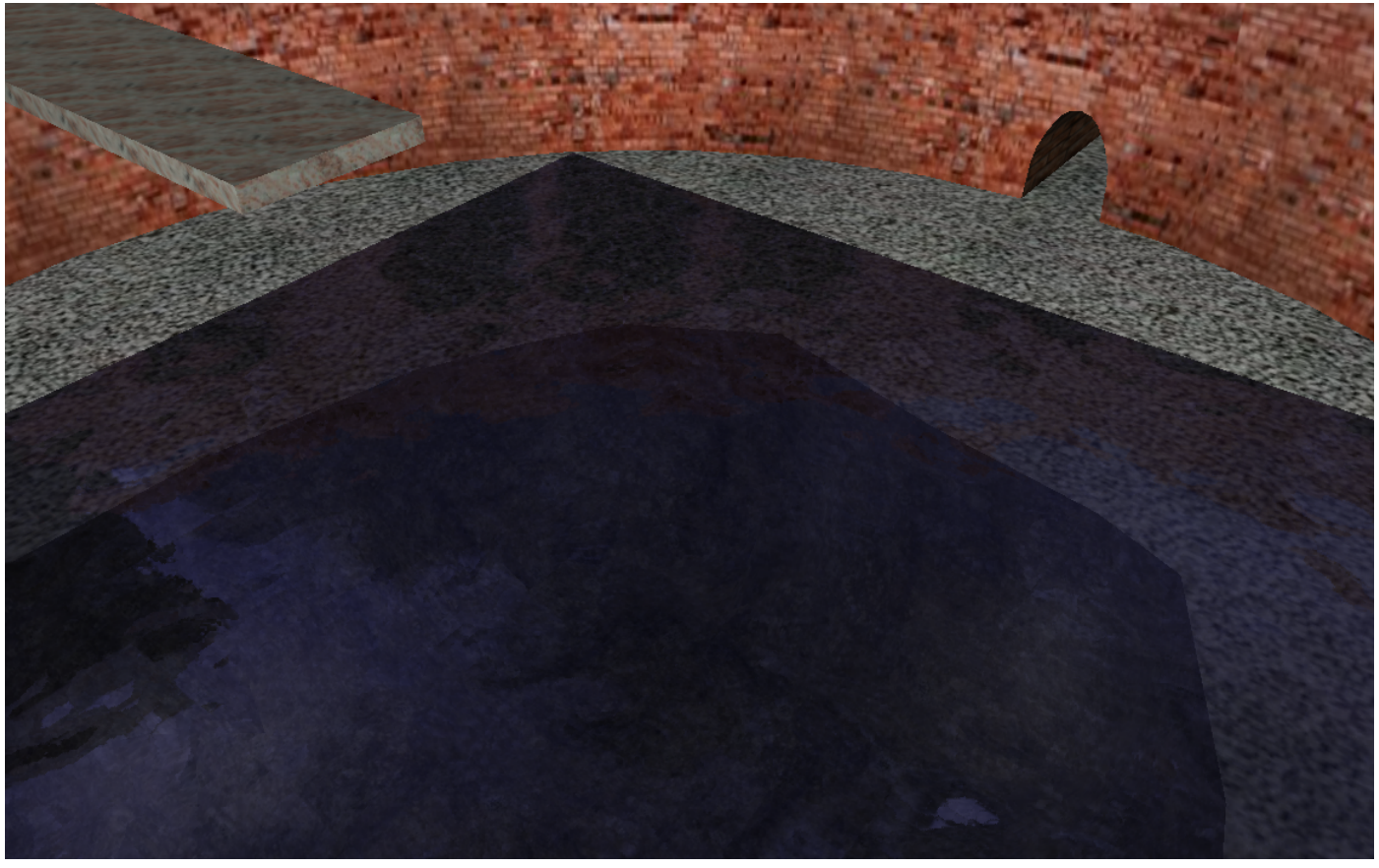
Krótki i działa na dowolnym modelu 3D!

Definiuj własne gniazda

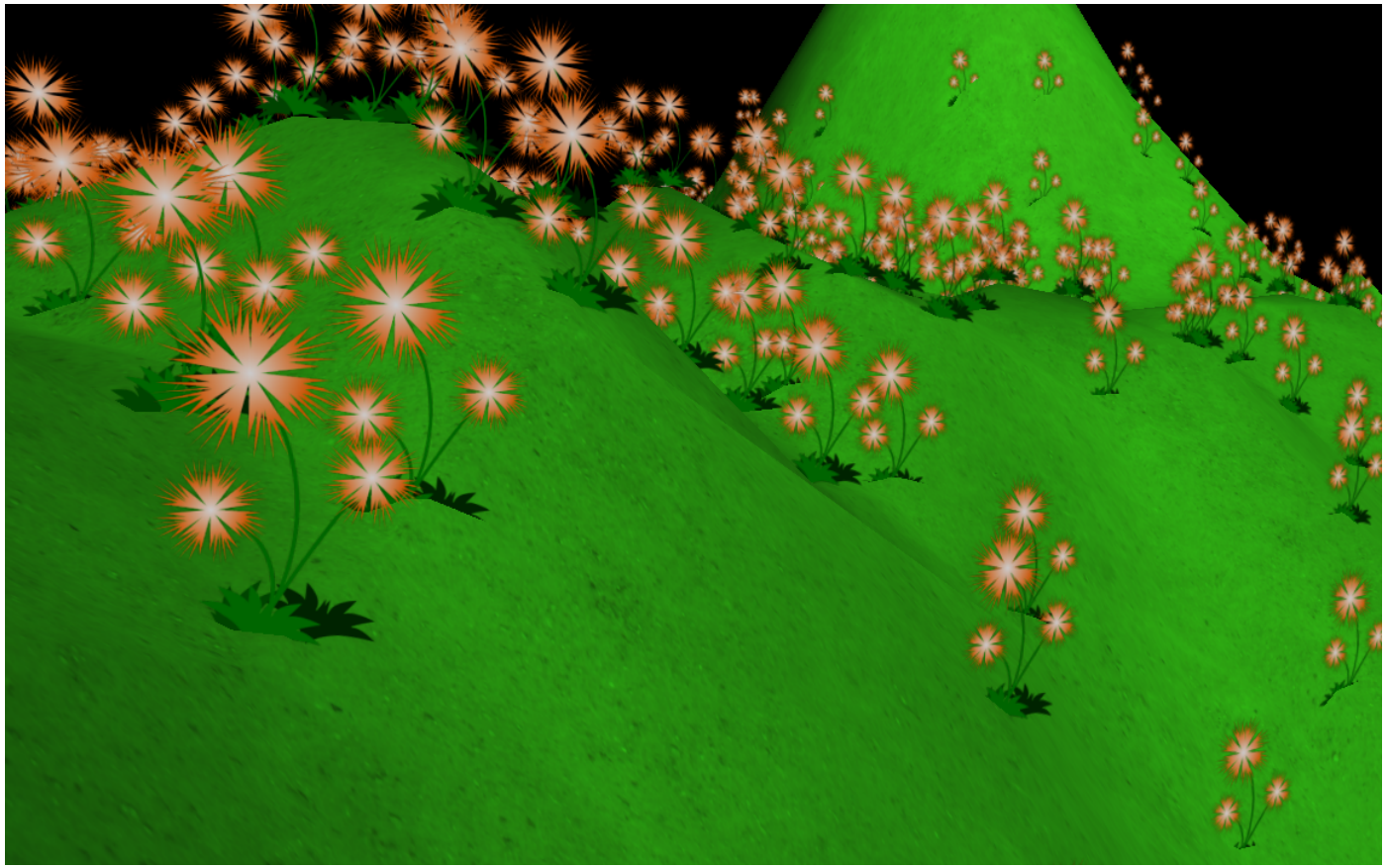
Cały system polega na tym że własne gniazda można definiować łatwo:

- Bazowy shader, i **każdy efekt**, może definiować gniazda do użycia przez następne efekty.
  Trywialne: magiczny komentarz `/* PLUG: ... */`.
- Magicznego komentarza można użyć wielokrotnie (żeby pozwolić na loop unrolling w shaderach).
- Magiczny komentarz `/* PLUG-DECLARATION */` (przydatny żeby można było zadeklarować wyższą wersję GLSL).
- Elegancko korzystamy z *separate compilation units* GLSL.
- Pomysł przenośny do innych języków shaderów.
- Dobrze dobrany domyślny zestaw gniazd, see linki na końcu i przykłady. Wbrew pozorom, nie trzeba 100 gniazd — istnieje kilka uniwersalnych gniazd które zaspokajają 90% potrzeb.

Przykłady końcowe - woda



Obraz: Woda, połączenie kilku efektów: generuj wektory normalne, transformuj je do eye space, połącz z reflection + refraction



Obraz: Animowane kwiatki: transformacja w object space zmieniana przez shader.

Plan

Wszystko jest zaimplementowane, open-source (LGPL), w moim silniku:

http://vrmlengine.sourceforge.net/

Instrukcje jak pobrać i oglądać przykładowe modele dotyczące komponowania shaderów:

http://vrmlengine.sourceforge.net/compositing_shaders.php

Dziękuję za uwagę!

Pytania?