Eurographics Symposium on Rendering 2010
Jason Lawrence and Marc Stamminger
(Guest Editors)

Volume 29 (2010), Number 4

Grammar-based Encoding of Facades - Additional Material

Simon Haegler[1], Peter Wonka[3], Stefan Müller Arisona[1,2], Luc Van Gool[1,4], Pascal Müller[2]

[1]ETH Zurich; [2]Procedural Inc.; [3]Arizona State University; [4]KU Leuven

Alternative Rule Evaluation

In principle, *F-shade* can be compiled to any shading language, e.g. GLSL, Nvidia's Cg, Microsoft's HLSL or offline languages like mental image's MetaSL, and Pixar's RSL. Prior to the CUDA implementation described in the main text, we tested a different strategy to implement *F-shade* in a GLSL fragment shader. The idea is to compile the grammar into a shading program itself so that each rule of the grammar is translated into one function in the shader with only a very small number of flow-control statements. Such an automatically generated (fragment) shader would consist of two parts: (1) a constant library part where the rule operations like `Split`, `Repeat`, etc are implemented and (2) the rules themselves translated into GLSL functions, where the successors are converted into direct function calls. The method works well and is very fast, *but*: the critical drawback of this method is the limitation of shading programs in length (our tests showed 65k assembler instructions). Due to the missing call stack on current GPUs all function calls are in-lined and the resulting shaders quickly get too long. We only managed to represent a couple of buildings with this method. A similar implementation in CUDA would suffer from the same limitation, although less severe.

We will use selected examples to explain the shader implementation. The header shows the variables used in the grammar. The *s*, *t* sampling position is given as a varying parameter computed by the rasterizer.

```
// globals:
const vec2 st = vec2(gl_TexCoord[0]);
vec4 scope = vec4(0, 0, 1, 1);
vec4 txScope = vec4(0, 0, 1, 1);
```

Every rule instance will be translated into one unique function call, however there are only nine different grammar rules. More specifically, this means that every one of the nine rules has a rule template that is called once for each time the rule exists with different parameters. The main objective of the function call is to modify the scope parameters. The only exception is the `Split` rule that returns an integer and requires a subsequent flow control operation to select a successor. For example, the following code shows the implementation for a repeat split. A repeat split takes a rectangular scope and subdivides it in as many elements of size `length` as there is space along the axis `a`.

```
void repeat(int axis, float length) {
   scope[a+2] *= length;
   float n;
   n = floor((st[a]-scope[a])/scope[a+2]);
   scope[axis] += n * scope[axis+2];
}
```

This building block can now be used for all repeat rules. We also show the code for the texture look-up function because it demonstrates how the scope and the texture scope values are finally used:

```
vec4 lookupTex(int texture) {
   float tx = (st[0]-txScope[0])/txScope[2];
   float ty = (st[1]-txScope[1])/txScope[3];
   return texture2D(texture, vec2(tx, ty));
}
```

Now, we can see how rules 1, 2, and 3 from the example grammar used in the main text is translated into shader code. In the `Facade()` function the generated GLSL code needs a flow control statement because of the split rule in the grammar. Also note how repeat is used to translate the `FloorsTex` rule. The repeat function is called to adjust the scope before calling the successor function.

```
vec4 Facade() {
 int i = split3(1,0.15,0.95);
 if (i == 0) return Groundfloor();
 else if (i == 1) return Floors();
 else return Cornice();
}

vec4 Floors() {
 trafoTex(0,0,0.1,0.05);
 return FloorsTex();
}

vec4 FloorsTex() {
 repeat(1,0.2);
 Floor();
}
```