Optimizing Shader Programs

The goal of shader optimizations is to increase its throughput (vertices/sec, fragments/sec) without impacting visual appearance.

Surface Effects and Combinations

Draw Objects Sorted by Texture

Spherical Environment mapping for static surface reflections is supported by Candera. The texture coordinate of the Spheremap will be calculated dynamically in the Shader depending on the position of the camera. Furthermore the spheremap texture can still be mixed with additional textures (multitexturing) like specular maps for shiny surfaces.

Consider using textures for storing (intermediate) results of expensive shader calculations. Texture lookups might be faster than arithmetic calculations.

Use Specialized Shaders

Texture Cache

Use Shaders that are customized to fit exactly your special situation.

Reference shaders are full featured and functional code examples to demonstrate the interoperability with Candera. They are not optimized for performance. A great performance improvement can be achieved by reducing the lighting calculation to support only the light sources that are actually used. They should be used for reference, but later be replaced by optimized and more specialized shaders.

Carefully Design your Lighting

Use Texture Compression

- Per-vertex lighting is less expensive than per-fragment lighting.
- Use as few lighting sources as possible (usually one light source is enough).
- Use simplified, tailored lighting calculations.
- Use light ranges to reduce the illumination area.
- Pre-calculate your lighting effects by putting them into textures.

Optimize Shader Program Coding

Avoid if-else Constructs in the Shader

The use of if-else statements for non-constant variables (attributes, varyings and local variables) should generally be minimized or avoided in shaders, as they might lead to synchronization penalties due to different processing times while parallel processing.

Reduce Calculations in the Shader Program

Reduce complex calculations in shader. Uniform calculations can be done outside of the shader. Attribute calculation must be done in the shader anyway.

For example rather give a pre-calculated model-view-projection matrix to the shader than combining the model view projection matrices within a shader, as each calculation is done for each vertex in the vertex shader. Candera pre-calculates uniforms set by Candera ShaderParamSetters automatically before given to the shader (like model-view-projection transformation, light vectors, reflection matrix, etc...).

Minimize the Number of (temporary) Variables

Remove Superfous Assignments

Consider that each assignment to a variable in a shader is executed for each vertex (in vertex shader) or fragment (in fragment shader). Thus, reduce the number of assignments to a right-minded minimum.

Minimize Variable Lifetime

Don't allocate variables across a large code range. Instead minimize their lifetime by scopes or by instantiation directly before usage.

use Lowest Precision Possible

Choose the lowest precision possible (without rendering artifacts) on vertices, normal, color and texture coordinates. Lower precision data requires less bandwidth to be transferred from VRAM to the graphics chip.

The shader performs most of the arithmetic calculations much faster in mediump than in highp precision. When using mediump precision for vertex attributes, make sure the data in memory is stored in half-float format as well. Otherwise the driver has to convert the precision every time the

object is drawn. Mediump precision should be the default precision in the fragment shader and for varyings.

Perform Vector Based Operations

If possible, make use of vector based operations like in the following example.

```
    vec2 x1 = (in1 + in1) / 2;
    vec2 x2 = (in2 + in2) / 2;
```

Better:

```
    vec4 x.xy = in1;
    x.zw = in2;
    x = (x+x) * 0.5;
```

Use Multiplication instead of Division

Better rephrase division by x as multiplication by 1.0/x, as this can be computed faster, generally.