Mesa's GLSL compiler

Eric Anholt

What is GLSL?

- C-like language operating on vector types
- OpenGL program gives the library a source code string
- GLSL compiler compiles it for the GPU to execute
- Used in vertex shading
  - Scale/translate/etc. model data to world space
  - Calculate lighting parameters
- Used in fragment shading
  - Compute color from interpolated parameters and textures

What does it look like?

```
uniform mat4 mvp

void main()
{
   gl_Position = mvp * gl_Vertex;
}
```

```
attribute vec2 in_texcoords;
varying vec2 texcoords;
uniform mat4 mvp

void main()
{
   gl_Position = mvp * gl_Vertex;
   texcoords = in_texcoords;
}
```

```
uniform vec4 color;

void main()
{
   gl_FragColor = color;
}
```

```
varying vec2 texcoords;
uniform sampler2D tex;

void main()
{
   gl_FragColor = texture2D(tex, texcoords);
}
```

It gets worse

```
#version 120

uniform vec3 light_eye;
varying vec2 texcoord;
varying vec3 light_surf;
varying vec3 eye_surf;
varying vec3 tangent_surf;
varying vec4 shadow_coords;
uniform mat4 mvp, mv, light_mvp;

void main()
{
    mat3 mv3 = mat3(mv);
    vec3 t = (mv3 * gl_MultiTexCoord1.xyz);
    vec3 n = (mv3 * gl_Normal);

    gl_Position = mvp * gl_Vertex;

    mat3 tbn = mat3(t,
            cross(n, t),
            n
            );

    vec3 vertex_eye = vec3(mv * gl_Vertex);
    shadow_coords = light_mvp * gl_Vertex;

    texcoord = gl_MultiTexCoord0.xy;
    light_surf = normalize((light_eye - vertex_eye) * tbn);
    eye_surf = normalize((-vertex_eye) * tbn);
    tangent_surf = gl_MultiTexCoord1.xyz * tbn;
}
```

```
void main()
{
        vec3 l = normalize(light_surf);
        vec3 v = normalize(eye_surf);
        vec3 h = normalize(l + v);
        vec3 t = normalize(tangent_surf);
        vec3 n = texture2D(normal_sampler, texcoord).xyz *
2 – 1;
        float n_dot_l = dot(n, l);
        float n_dot_v = dot(n, v);
        float n_dot_h = dot(n, h);
        float v_dot_h = dot(v, h);
        float cos2_alpha = n_dot_h * n_dot_h;
        float tan2_alpha = (1 - cos2_alpha) / cos2_alpha;
        float cos_phi = dot(normalize(t.xy),
normalize(h.xy));

        float cos2_phi_over_m2 = (cos_phi * cos_phi) *
ward_mm_inv;
        float sin2_phi_over_n2 = (1 - cos_phi * cos_phi) *
ward_nn_inv;
        D = exp(-tan2_alpha * (cos2_phi_over_m2 +
sin2_phi_over_n2));
        Rs = 2 * schlick_fresnel(n_dot_l) * D *
            inversesqrt(n_dot_l * n_dot_v) * ward_mn_inv;
        Rs *= s;

        gl_FragColor = max(0, n_dot_l) *
            step(0, n_dot_v) *
            vec4(material_color.xyz *
                 ((Rd * d + Rs) * Ii * shadow),
                 material_color.w);
}
```

We need a compiler

- Not just parsing into a syntax tree
- We want actual optimization

Why it's easy

- Compiler techniques are extremely well known
- lex, yacc handle some irritating parts
- Programs are short
- No such thing as memory
- No such thing as pointers

Why it's hard

- Most GPUs don’t look like CPUs
- vec4 as the basic datatype
- write masks on register destinations
- source swizzles (channel moves, replacement with constants)
- Many GPUs don’t have things like “if” or “loop”

Write masks

- Optimization wants to know “where does this value come from?”
- Easy to answer with scalar values: the last thing to write to it
- What is the answer for vectors?

```
varying vec2 texcoords;
uniform sampler2D tex;

void main()
{
   vec4 color = texture2D(tex, texcoords);
   color.rgb = mix(color.rgb, vec3(0.633), 0.2);

   gl_FragColor = color;
}
```

There are two answers

- Deciding whether to treat vectors as vectors depends on GPU
  - "AOS" is having one register with the whole vec4 in it.

reg0	x0	y0	z0	w0
reg1	x1	y1	z1	w1
reg2	x2	y2	z2	w2
reg3	x3	y3	z3	w3

  - "SOA" is having 4 registers for a vec4.

reg0	x0	x1	x2	x3
reg1	y0	y1	y2	y3
reg2	z0	z1	z2	z3
reg3	w0	w1	w2	w3

SOA vs AOS

- 965 vertex is AOS
- 965 fragment is SOA
- 915 is AOS
- r200 is AOS
- r300/r500 is AOS
- r700 is AOS
- nv40 is AOS
- nv50 is SOA
- nvc is SOA

GPU limitations: Flow control

- GPUs don't do arbitrary flow control
- As of ~6 years ago, GPUs did no flow control
- GLSL requires support for loops and if statements
- Tell the loop unroller to unroll everything
- Replace if..else..endif blocks with conditional moves

GPU limitations: Array access

- Some GPUs just don't do this
- GLSL requires that you do
- Allocate a bunch of registers, do conditional moves
  - Does this sound familiar?

GPU limitations: Instruction count

- Old GPUs can often do just a few instructions
  - 915: 64 ALU, 32 texturing
  - r200 vertex: 128 instructions
  - r300 vertex: 256 instructions
  - r500 vertex: 1024 instructions
- If we fail at optimizing, it's worse than running slow

GPU limitations: registers and memory

- Until recently, no memory access at all
  - 915: 16 temporary registers
  - r200 vertex: 12 temporary registers
  - r300 vertex: 32 temporary registers
- Register allocation is a big deal
  - If you've got no memory access, no spilling allowed
  - Even if you have memory access, spilling is expensive
    - One shader spilling reduced Lightsmark performance 50% on 965

GLSL advantages

- Not IEEE floats
- Almost no guarantees about your math.
  - 1/1/x == x
  - 2.0 * x * 0.5 == x
  - sin() might be sin(), might be a small-order polynomial.

Conclusion

- New compiler is in place in Mesa 7.9
  - i915 got GLSL support
- New native codegen for 965 fragment shader in Mesa 7.10
  - nexuiz 20% faster than in Mesa 7.8
- Most programs generate good-looking code
- Still work to do to optimize some programs
- Still need native codegen for other GPUs
- Still need native codegen for the CPU