Programming in GLSL is not programming in C

Some traps, performances issues, compilation failures
Some recommended computation models

ref: shadertoyunofficial blog
Table of Content

- Various important details
  - misc
  - compilation

- What happens to your code at compilation

- What happen to your code at run-time

→ Some recommendations

- Some more details

Disclaimer: bias glsl, wGl, fragment [ shadertoy :-D ]
Various important details: misc

- Many built-in funcs:
  - geometry: reflect, refract, length, dot, cross, normalize…  [ + clamp, mix, smoothstep… ]
  - matrices: ops on n×m up to 4
  - textures, interp, MIPmap  [ but some bugs ]

- Loose specs:
  - loose IEEE:
    - NaN not treated by every built-in funcs ( min, max, clamp, smoothstep… )
    - denormalized
    - NaN and Inf for const vs non const
  - loose portability

- Think procedural ( pull/Eulerian rather than push/Lagrangian :-) ) :
  - loop on regular items
    → guess which items can cover the pixel
    → rely on mod(); distance function  [ example: crowded ]
Various important details: misc

- Many built-in funcs:
  - geometry: reflect, refract, length, dot, cross, normalize...  
    [ + clamp, mix, smoothstep... ]
  - matrices: ops on n×m up to 4
  - textures, interp, MIPmap
    [ but some bugs ]

- Loose specs:
  - loose IEEE maths:
    [ IEEE 754 floats ]
    - NaN not treated by every built-in funcs ( min, max, clamp, smoothstep... )
    - denormalized numbers
  - NaN and Inf for const vs non const
  - loose portability

- Think procedural ( pull/Eulerian rather than push/Lagrangian :-) ) :
  - loop on regular items
    → guess which items can cover the pixel
    → rely on mod(); distance function ( example: crowded )
Various important details: misc

- Many built-in funcs:
  - geometry: reflect, refract, length, dot, cross, normalize… [ + clamp, mix, smoothstep… ]
  - matrices: ops on n×m up to 4
  - textures, interp, MIPmap [ but some bugs ]

- Loose specs:
  - loose IEEE maths: [ IEEE 754 floats ]
    - NaN not treated by every built-in funcs ( min, max, clamp, smoothstep… )
    - denormalized numbers
  - NaN and Inf for const vs non const
  - loose portability

- Think procedural ( pull/Eulerian rather than push/Lagrangian :-( ) :
    - loop on regular items
      → guess which items can cover the pixel
      → rely on mod(); distance function [ example , 2 , crowded ]
Various important details: compilation

- Many language targets:
  - openGL vs openGL ES vs webGL vs Vulkan vs HLSL
    [ WebGL 2.0 ~ OpenGL ES 3.0 ~ OpenGL 3.3 + 4.2 ]
  - version
  - extensions ( + core vs legacy )
  - `get_program_binary()` vs
    `> cgc bug.glsl -ogles -profile fp40`
    [ nvidia-cg-toolkit ]

- Compilation steps:
  - web: Angle patches (browser dependent)
  - web: possibly, transpilation to HLSL/D3D (version browser dependent)
    or choice of openGl target language
  - GLSL compiled into ARB ( in driver )
    HLSL compile into intermediate (in D3D) then ARB ( in driver )
    [ maybe ? ]
  - ARB compiled into PTX ( on GPU )
  - rewriting (for bug/perf fixes + optimizations) occurs at every steps

NB: Bug reports: https://bugs.chromium.org community ultra-efficient
[ mix of Nvidia/Intel/Microsoft/Angle/Chrome coders ]
Various important details: compilation

- Many language targets:
  - openGL vs openGL ES vs webGL vs Vulkan vs HLSL  
    [WebGL 2.0 ~ OpenGL ES 3.0 ~ OpenGL 3.3 + 4.2]
  - version
  - extensions ( + core vs legacy )
  - get_program_binary() vs > cgc bug.glsl -ogles -profile fp40 [nvidia-cg-toolkit]

- Compilation steps:
  - web: Angle patches (browser dependent)
  - web: possibly, transpilation to HLSL/D3D (D3D version browser dependent)
    or choice of openGl target language (desktop vs ES vs wGl)
  - GLSL compiled into ARB (in driver)
    HLSL compile into intermediate (in D3D) then ARB (in driver)  
    [maybe ?]
  - ARB compiled into SAS (on GPU)  
    [Cuda: PTX then SAS]
  - code rewriting occurs at every steps (for bug/perf fixes + optimizations)

NB: Bug reports: https://bugs.chromium.org community ultra-efficient [mix of Nvidia/Intel/Microsoft/Angle/Chrome coders]
What happen at compilation

- no true functions → inlined
  [ no stack, no recursivity, macro-like ]
- loops → optimizer unroll if it can
  [ even if gives stupidly long code or compile time or endless ]
- branches → both might be evaluated

while ( marching ray, up to 100 steps ) {
    p = next ray sample
    if hit(p) {
        eval N(p); eval material(p);
        I = shadow(p,L)*color(L);
        outColor = shading(N,material,I);
        break;
    }
}

hit(p); // compute intersection against N shapes parts. + possible proceduralism.
N(p);   // finite difference on shape [ hopefully not doing FDiff(hit(p)) ]
shadow(p,L); // march shadow ray (loop, hit, material... )
material(p); // procedural noise, textures fetches, ...
What happen at compilation

- no true functions → inlined
- loops → optimizer unroll if it can
- branches → both might be evaluated

while ( marching ray, up to 100 steps ) {
    p = next ray sample;
    if hit(p) {
        eval N(p); eval material(p);
        I = shadow(p,L)*color(L);
        outColor = shading(N,material,I);
        break;
    }
}

hit(p); // compute intersection against N shapes parts + possible proceduralism.
N(p); // finite difference on shape [ hopefully not doing FDiff(hit(p)) ]
shadow(p,L); // march shadow ray (loop, hit, material... )
material(p); // proceduralism, noise, textures fetches, ...
What happen at run-time

Conditional branching vs divergence (SIMD)

- **Facts:** divergence in warp → both branches evaluated for all
  - big then/else blocks → (code length), runtime length
  - loop + if (end) break → can give messy code
  - dFdx, dFdy, fwidth undetermined, or 0, or rand...
  - texture LOD undetermined, or 0, or rand... or might hide 4 x code duplicate → manual LOD
  - dF, LOD: pushed out of early exited loop won't save. True deferred will.

- **Myths:**
  - In many situation, unlikely divergence in warps (are just 32 pixels)
  - If process in branch is small, no problem
  - `mix(expr0, expr1, float(cond))` is just counterproductive! [but `mix(v0,v1,bvec)` is ok]
  - `?:` compiles just like shorts if else [still some doubt how chains of `?:?:?:` are evaluated]
What happen at run-time

Conditional branching vs divergence (SIMD)

- Facts: divergence in warp → both branches evaluated for all
  - big then/else blocks → (code length), runtime length
  - loop + if (end) break → can give messy code
  - dFdx, dFdy, fwidth undetermined, or 0, or rand...
  - texture LOD undetermined, or 0, or rand... or might hide 4 x code duplicate → manual LOD
  - dF, LOD: pushed out of early exited loop won’t save. True deferred will.

- Myths: ‘if’ is not Evil per se
  - In many situation, unlikely divergence in warps (are just 32 pixels)
  - If branch block is small, no problem
    mix( expr0, expr1, float(cond) ) is just counterproductive!
  - magic thinking: ?: compiles just like shorts if else
    [ still some doubt how ?:?:?: is evaluated ]
    [ but mix(v0,v1,bvec) is ok ]
    [ but dithered code is evil ]
Recommendations

- Deferred heavy processing out of loops:
  replace if (end_condition) { process; break; }
  with if (end_condition) { set_parameters; break; }

- Deferred heavy processing out of branches:
  replace ...else if ( cond_N ) do_action(params);
  with ...else if ( cond_N ) set_parameters;

- Specialize functions, or use branches inside only if triggered by const params:
  - worst case would be shape( P, [not const] kind, params )
  - shadows: loop, hit, material should be simpler

- Forbid unrolling when stupid:
  for (int i=0; i<N+min(0,positive not const); i++)

- Special flags and qualifiers:
  #pragma optimize(off), varying, coherent, volatile, restrict, readonly, writeonly... [out of my competence]
→ Recommendations

- Deferred heavy processing out of loops:
  replace if (end_condition) { process; break; }
  with if (end_condition) { set_parameters; break; }

- Deferred heavy processing out of conditional branches:
  replace ...else if (cond_N) do_action(params);
  with ...else if (cond_N) set_parameters;

- Specialize functions, or use branches inside only if triggered by const params:
  - worst case would be shape( P, [not const] kind, params )
  - shadows: loop, hit, material should be simpler

- Forbid unrolling when stupid:
  for (int i=0; i<N+min(0, positive not const); i++)

- Special flags and qualifiers:
  [ out of my competence ]
  #pragma optimize(off), varying, coherent, volatile, restrict, readonly, writeonly...
Recommendations

- Deferred heavy processing out of loops:
  replace if (end_condition) { process; break; }
  with if (end_condition) { set_parameters; break; }

- Deferred heavy processing out of conditional branches:
  replace ...else if ( cond_N ) do_action(params);
  with ...else if ( cond_N ) set_parameters;

- Specialize functions, or use branches inside only if triggered by const params:
  - worst case would be shape( P, [not const] kind, params )
  - shadows: loop, hit, material should be simpler

- Forbid unrolling when stupid:
  for (int i=0; i<N+min(0,positive not const); i++)

- Special flags and qualifiers: [ out of my competence ]
  #pragma optimize(off), varying, coherent, volatile, restrict, readonly, writeonly...
→ Recommendations

- Deferred heavy processing out of loops:
  
  replace `if (end_condition) { process; break; }`
  with `if (end_condition) { set_parameters; break; }`

- Deferred heavy processing out of conditional branches:
  
  replace `...else if ( cond_N ) do_action(params);`
  with `...else if ( cond_N ) set_parameters;`

- Specialize functions, or use branches inside only if triggered by const params:
  - worst case would be `shape( P, [not const] kind, params )`
  - shadows: loop, hit, material should be simpler

- Forbid unrolling when stupid:
  
  `for (int i=0; i<N+min(0,positive not const); i++)`

- Special flags and qualifiers: [out of my competence]
  
  `#pragma optimize(off), varying, coherent, volatile, restrict, readonly, writeonly...`
Recommendations

- Deferred heavy processing out of loops:
  replace if (end_condition) { process; break; } with if (end_condition) { set_parameters; break; }

- Deferred heavy processing out of conditional branches:
  replace ...else if ( cond_N ) do_action(params); with ...else if ( cond_N ) set_parameters;

- Specialize functions, or use branches inside only if triggered by const params:
  - worst case would be shape( P, [not const] kind, params )
  - shadows: loop, hit, material should be simpler

- Forbid unrolling when stupid:
  for (int i=0; i<N+min(0,positive not const); i++)

- Special flags and qualifiers :
  [ out of my competence ]
  #pragma optimize(off), varying, coherent, volatile, restrict, readonly, writeonly... [?]
→ Recommendations

- Calculus model: pipelined

  loop ( march ray ) → hit point
  compute N, material
  loop ( march shadow ) → I
  compute shading

- Calculus model: deferred

  pass 1 → storage
  storage → pass2

[ added gift: better for registers ]
→ Recommendations

- Calculus model: pipelined

  
  loop ( march ray ) → hit point
  compute N, material
  loop ( march shadow ) → I
  compute shading

- Calculus model: deferred

  [ added gifts: better for registers, dFdx ]

  pass 1 → storage
  storage → pass2
→ Recommendations

Calculus model: pipelined

→ **GigaVoxels**: octree with voxel grids in not empty nodes

- **bad**: [ warp might get divergent, even if all grids ]

  ```
  while (march ray through octree) {
    if (grid) march_grid();
  }
  ```

- **good**:

  ```
  while (not finished) {
    step 1 octree node;
    if (grid) march_grid();
  }
  ```
Some more details: optimizer

- pull from output:
  - unused code removed (comprising unused vec4 components)
  - might unmap uniforms
- some pattern detection, but…
  - sqrt, invsqrt, length, normalize…
- test 1:
  - factor expr(uniform) out of loop; recognize *0
  - don’t detect empty loop
- test 2:
  - detect empty loop j
  - don’t factor expr(i) out of loop j
- test 3: recognize expr already calculated
  - only if it was end result: expr+1 not help expr-1
  - still, 1.*expr-0. seen as expr

`get_program_binary()` vs `cgc bug.glsl -gles -profile fp40`
Some more details

- multiple compilations
  - compiler tries multiple optimization strategies
  - at runtime: perf increases with time!

- no branch prediction
  - no Spectre exploit on GPU :-)
  - order tests by decreasing probability

- generalization
  - to Cuda? OpenCL?
  - to C?