Getting There First:

OpenGL ES 3.1 and Intel Extensions in Fractal Combat X

Jon Kennedy, Egor Yusov,
Davide Pasca

March 04, 2015

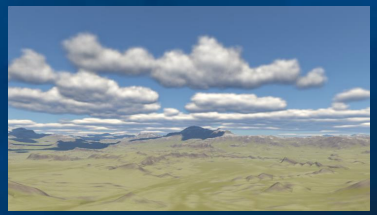
Legal

Presenter Bios

- Jon Kennedy, Intel
  - Senior Graphics Engineer in the Advanced Technology Group
- Egor Yusov, Intel
  - Senior Graphics Engineer in the Visual Computing Engineering Group
- Davide Pasca, OYK Games
  - Lead Developer of Fractal Combat X

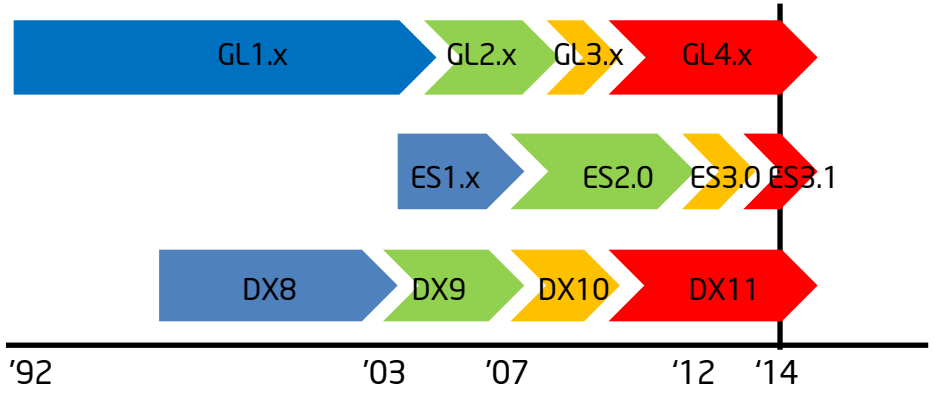
Agenda

- OpenGL ES 3.1 Overview
  - Including the Android Extension Pack
- Explore the Intel Extensions
  - Tessellation,
  - Geometry Shaders,
  - Fragment Shader Ordering
- Working with x86 and OpenGL ES 3.1 in Fractal Combat X
  - Compiler Setup
  - SSE
  - Porting from OpenGL ES 2.0 to OpenGL ES 3.1
  - Intel extension usage



OpenGL ES 3.1 Overview

- Intel was the first and only IHV demonstrating OpenGL ES 3.1 games at GDC 2014
  - OpenGL ES 3.1 is supported on Intel's Bay Trail and Cherry Trail mobile platforms
- Android 5.0 natively supports OpenGL ES 3.1
  - Intel offers support in Android 4.4
- OpenGL ES 3.1 + AEP brings parity with desktop
  - Full desktop rendering pipelines can now be run on mobile platforms



What is New in OpenGL ES 3.1?

- **Shader Stages**
  - Compute Shaders
- **Shader I/O**
  - Shader Image Load Store
  - Shader Storage Buffer Objects
- **Shader Synchronisation**
  - Atomic Counters and Memory Ops
  - INTEL: Fragment Shader Ordering
- **API Helpers**
  - Indirect Draw Commands
  - Framebuffer No Attachments
  - Program Interface Query
  - Separate Shader Objects
  - Vertex Attribute Binding
- **Textures**
  - Texture Storage Multisample
- **GLSL**
  - Array of Arrays
  - Explicit Uniform Locations
  - Shader Bitfield Operation
  - Shader Layout Binding
  - Texture Gather
  - Stencil Texturing

Core

Intel Extension

What is the Android Extension Pack?

- Google have worked to roll up ~20 extensions to form the extension pack
  - https://www.khronos.org/registry/gles/extensions/ANDROID/ANDROID_extension_pack_es31a.txt
- It provides a feature level above core OpenGL ES 3.1
- It is supported natively in Android 5.0
- Some hardware has partial AEP support
  - For best market penetration
  - Do not
    - Add the Android Extension String to the Android manifest
    - Look for the extension string for the extension pack in your app
  - Do
    - Look for the individual extensions you plan to use
    - Use fall-back paths where the extensions are not supported

OpenGL ES 3.1 + AEP

- **Shader Stages**
  - Compute Shaders
  - EXT: Tessellation Shaders
  - EXT: Geometry Shaders
- **Shader I/O**
  - Shader Image Load Store
  - Shader Storage Buffer Objects
- **Shader Synchronisation**
  - Atomic Counters and Memory Ops
  - OES: Shader Image Atomics
  - INTEL: Fragment Shader Ordering
- **API Helpers**
  - Indirect Draw Commands
  - Framebuffer No Attachments
  - Program Interface Query
  - Separate Shader Objects
  - Vertex Attribute Binding
  - KHR: Debug
  - EXT: Primitive Bounding Box
  - EXT: Copy Image
- **Textures**
  - Texture Storage Multisample
  - OES: Texture Stencil 8
  - KHR: ASTC (LDR)
  - EXT: Texture Storage MS Array
  - EXT: Texture Border Clamp
  - EXT: Texture Buffer
  - EXT: Cube Map Array
  - EXT: sRGB Decode
- **Blending**
  - KHR: Blend Equation Advanced
- **GLSL**
  - Array of Arrays
  - Explicit Uniform Locations
  - Shader Bitfield Operation
  - Shader Layout Binding
  - Texture Gather
  - Stencil Texturing
- **GLSL Extensions**
  - OES: Shader Multisample Interpolation
  - OES: Sample Shading
  - OES: Sample Variables

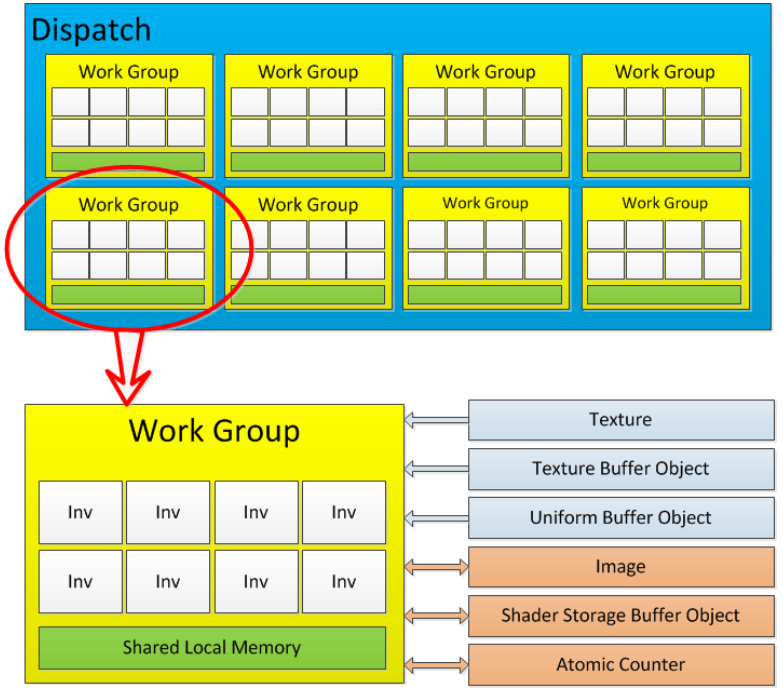
Core

Android Extension Pack

Intel Extension

Compute Shaders

- General-purpose computations on the GPU
  - Post-processing, physics, simulations, animation, etc.
  - Runs logically independent of the 3D pipeline
  - Dispatch 3 dimensional array of workgroups
    - Each workgroup has a 3 dimensional array of shader invocations
    - This is specified at shader compile time
- Each workgroup has access to shared local memory
  - Shader invocations within the workgroup share this memory
  - Requires synchronization
- Can directly access textures, images and buffer objects as well as atomic counters
- Caution
  - Difficult to ascertain the optimal work group and shader invocation sizes for the algorithm and data set
  - Different optimal sizes for different GPUs
  - If performance is important and if the algorithm does not require shared local memory, use a fragment shader



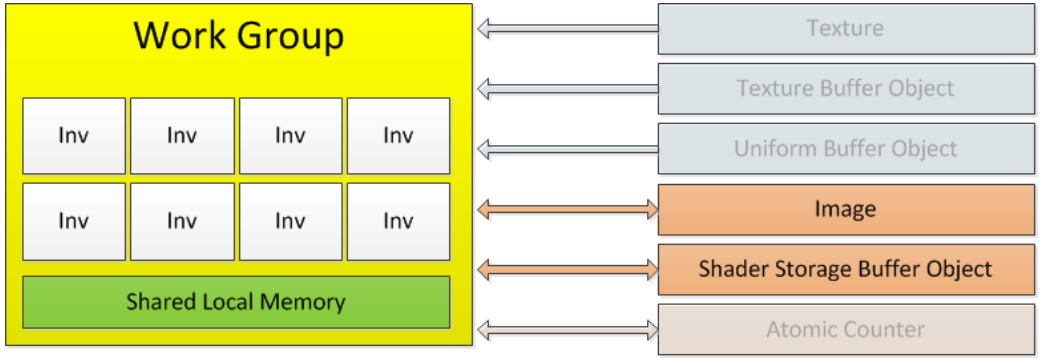
Compute Shader I/O

Shader Image Load/Store

- Random read/write access to a single level of a texture
- Atomic operations
- Texture must be immutable
  - glTexStorage*, not glTexImage*
  - Can be a buffer texture
- Specify read/write requirements at bind time

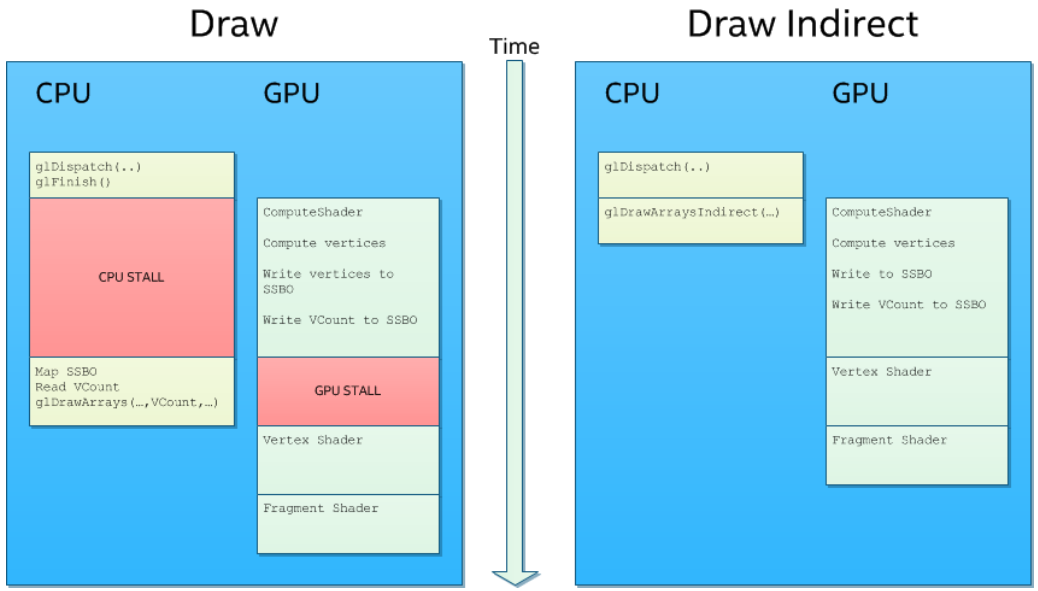
Shader Storage Buffer Objects

- Random read/write access to a buffer object
- Atomic Operations
- New bind point for buffer objects
  - GL_SHADER_STORAGE_BUFFER
- Used mainly in compute shaders, but supported in other stages



Indirect Draw Commands

- Draw call parameters sourced from a buffer object
  - Buffer object bound to GL_DRAW_INDIRECT_BUFFER
  - Offset into buffer object is passed into the draw call
- Avoids CPU/GPU Sync
- Useful for GPU generated drawcalls
  - Compute shader can generate vertices and generate draw call without feedback loop
- Supports :
  - glDrawElementsIndirect
  - glDrawArraysIndirect
  - glDispatchComputeIndirect



ASTC Texture Format

- Adaptive Scalable Texture Compression Format
- Supports multiple formats including RGBA, Normal XY or XYZ, LDR/HDR and 3D textures.
  - Not all hardware supports all features
- Fixed block size of 128-bits
- Variable block footprint from 4x4 to 12x12
- Bit-rate ranging from 8 bits to 0.89 bits per texel
- ASTC LDR supported on Intel Cherry Trail mobile platform

2 bits/pixel

3.56 bits/pixel

8 bits/pixel

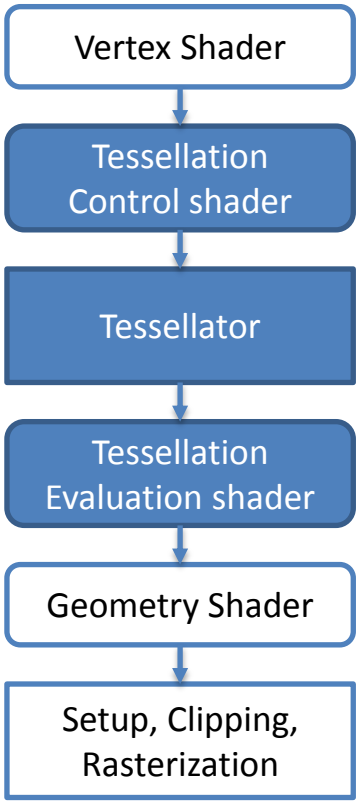
http://en.wikipedia.org/wiki/Adaptive_Scalable_Texture_Compression

Intel Extensions

- Tessellation
- Geometry Shaders
- Fragment Shader Ordering
- Available on Intel Mobile Platforms with/without the Android Extension Pack

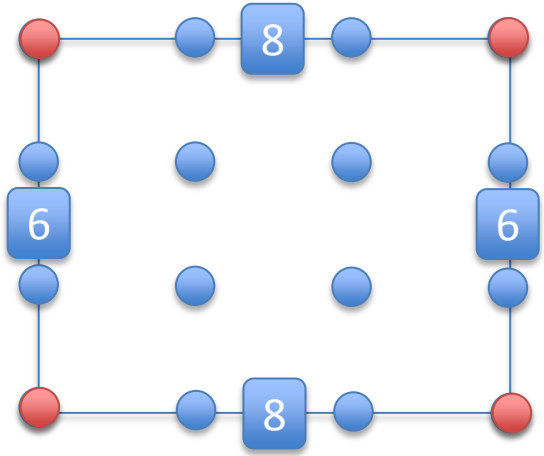
Tessellation

- Two new programmable stages and one new fixed-function stage to dynamically generate tessellation on the GPU
  - Tessellation control shader
  - Tessellator
  - Tessellation evaluation shader
- New primitive type: patch
  - General-purpose primitive containing from 1 to at least 32 vertices



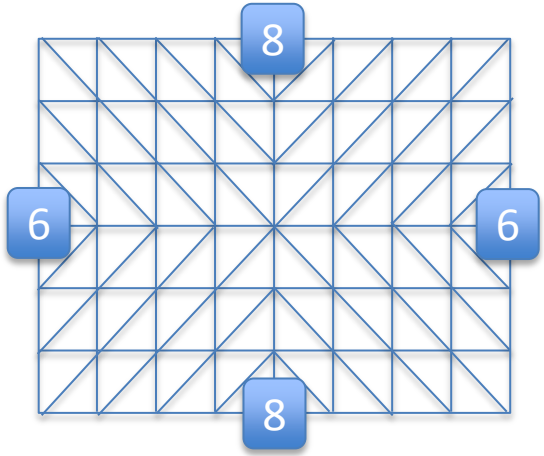
Tessellation Control Shader

- Performs any special transformations on the input data
  - Can change the number of vertices
  - Executed once per *output* vertex
  - Can share data between different executions processing the same patch
- Determines the amount of tessellation that a primitive should have



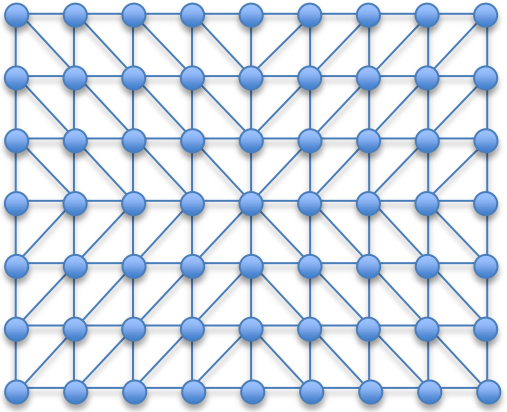
Tessellator

- Fixed-function stage
- Generates a set of new primitives using tessellation levels provided by the TCS
- Three patch types are supported:
  - Quads
  - Triangles
  - Isolines
- Only tessellation levels are used, any other control vertex data is ignored



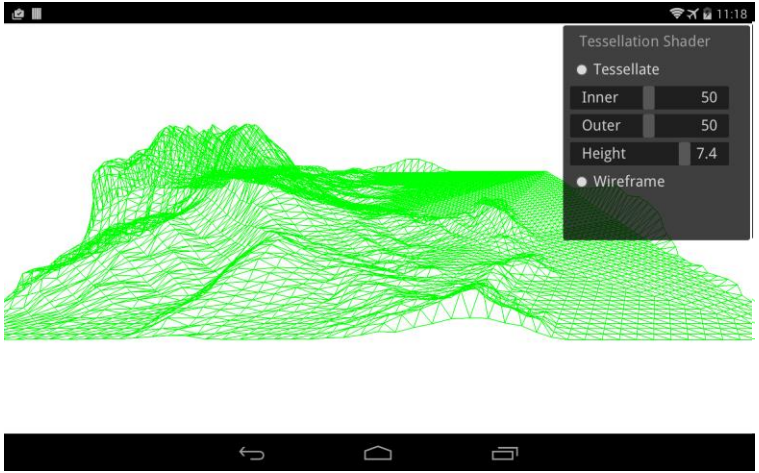
Tessellation Evaluation Shader

- Computes actual positions (and other attributes) for vertices generated by the tessellator
- Executed once per vertex in the patch tessellation
- Very similar to vertex shader



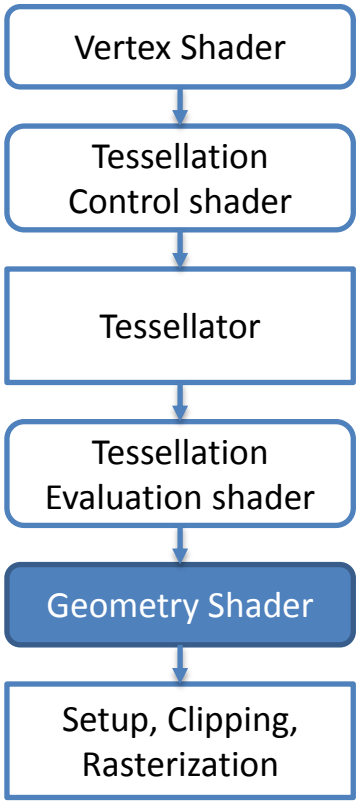
Tessellation: Use Cases

- Dynamic LOD systems
- Terrain rendering
- Displacement mapping
- Subdivision surfaces
- Hair, fur, grass
- Extension string:
  - GL_INTEL_tessellation
  - GL_EXT_tessellation



Geometry Shaders

- New optional programmable stage after VS (and tessellation if enabled)
- Operates on whole *primitives*
  - One primitive in
  - Zero, one or more primitives out
- Primitive types
  - Points
  - Lines
  - Triangles
- Optional adjacency information

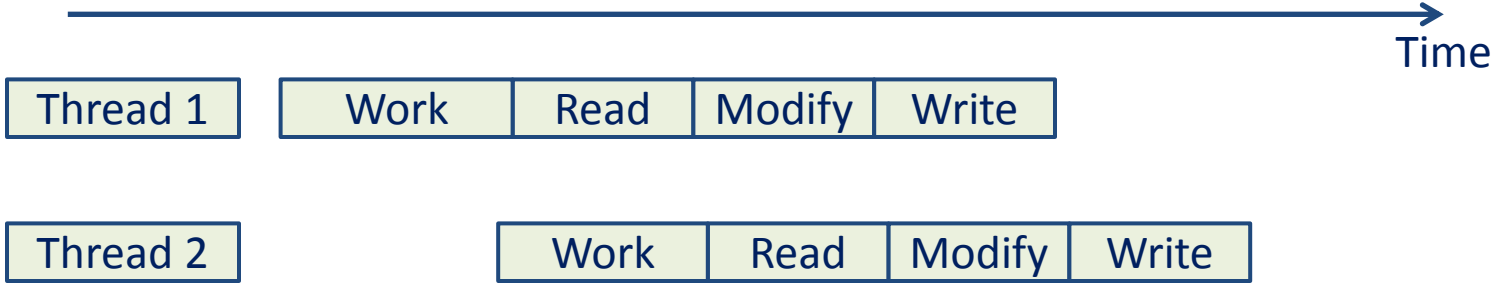


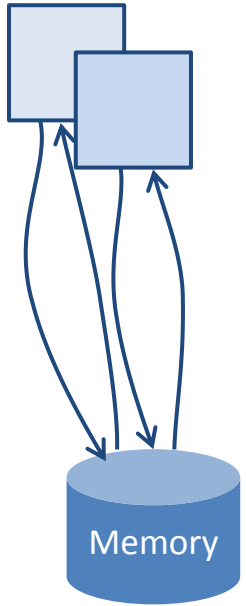
Geometry Shaders: Use Cases

- Particle generation
- Rendering to MRT
  - Rendering to cubemap faces
- Transform feedback
- NPR
- Shadow volume extrusion
- Extension string:
  - GL_EXT_geometry_shader
  - GL_INTEL_geometry_shader

Fragment Shader Ordering

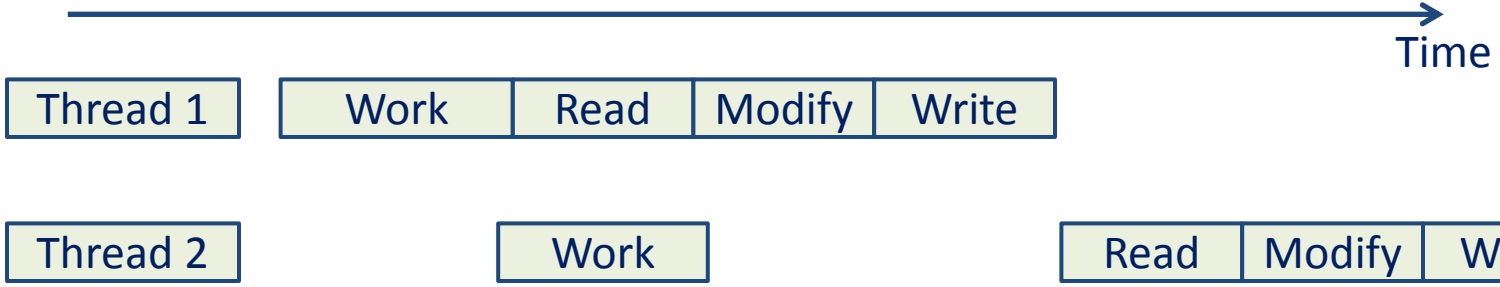
- OpenGL ES does not impose any ordering on the execution of the fragment shader
  - Ordering happens later at the blend stage
- If two threads read and modify the same memory in an SSBO or Image, the result is unpredictable





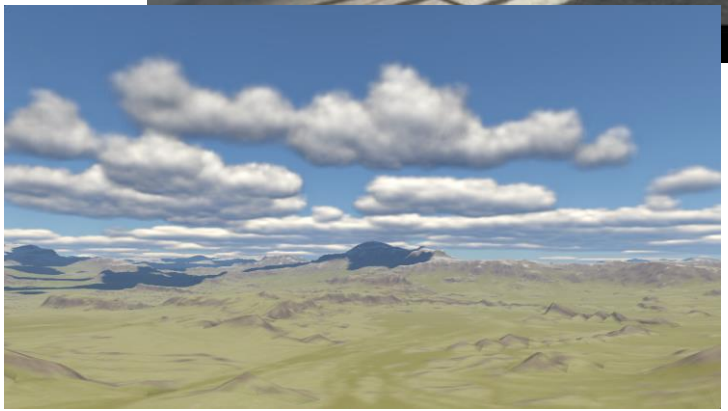
Fragment Shader Ordering

- Fragment Shader Ordering assures that
  - Read-modify-write operations are protected, i.e. no thread can read the memory before other thread finishes writing to it
  - All memory access operations happen in the same order in which primitives were submitted for rendering



Fragment Shader Ordering: Use Cases

- Order-independent transparency
- Custom blending
- Volumetric Shadow Maps
- Clouds
- Extension string:
  - GL_INTEL_fragment_shader_ordering



Fractal Combat X

Fractal Combat X: overview

- Arcade flight combat / shooter
- 1.5 M downloads (free-to-play)
- For iOS, Android, Android consoles
- Elevation map terrain with LOD
- Multi-platform in-house engine

Building for x86

- Already developing on Windows with Visual Studio (cross platform)
- Very straightforward to build “fat” binaries
- Early stability problems with previous NDKs
  - Auto-vectorization for x86 only working since gcc 4.9
  - NOTE: flags in 4.9 change!
    e.g. *–ftree-vectorize-verbose* becomes *-ftree-info-vec*
- Custom build based on Make, helped with switching compilers

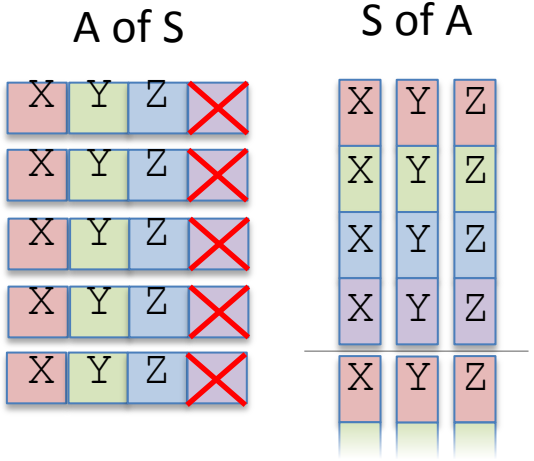
SIMD on mobile

- In super-computers in 70s, PC in 90s and mobile now
- Ideal for 3D graphics: matrix, vectors
- Standard for many years on desktop, console, iOS
- Not guaranteed on Android ARM
  - Check at run-time and pick different code path (we gave up)
- Standard on Android x86! (SSE4)
  - Compile-time setting

How to SIMD?

Array of Structs vs Struct of Arrays

- AoS (classical Vec4 class)
  - More intuitive, less efficient, *not* future proof (AVX has 8 floats per register)
- SoA
  - More flexible, more efficient
  - But manual for-loop split is not fun



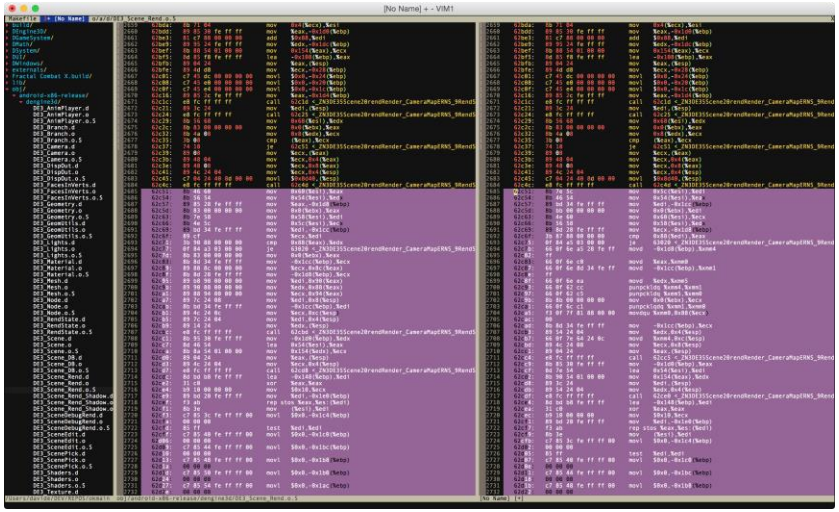
Already wasteful for 3D with 128bit, 4-floats registers

How to SIMD? (cont.)

- Solution: use Auto-vectorization
- Benefits of SoA without the details
- Still, need to follow best practices
- Possibly check assembly output

```
for (int r=0; r < 4; ++r)
{
  for (int c=0; c < 4; ++c)
  {
    float sum = 0;
    for (int i=0; i < 4; ++i)
    {
      sum += m1.mij(r,i) * m2.mij(i,c);
    }
    mout.mij(r,c) = sum;
  }
}
```

-fopt-info-vec ***objdump –S*** ***diff***



OpenGL ES 3: API setup

- Use NDK-provided **gl3stubInit()** to setup new function pointers, at run-time *if* ES 3 is available
- Do ***not*** rely on **glGetString(GL_VERSION)**
  - Some drivers report highest version supported, regardless of the context version
- Instead, to get **major** version, use:

```
eglQueryContext( dpy, ctx,
                 EGL_CONTEXT_CLIENT_VERSION,
                 &major )
```

ES 2.0 to ES 3.1: Shaders

- Annoying shaders changes, when targeting **both** ES 2 and ES 3
- ***attribute*** becomes ***in***
- ***varying*** becomes ***in*** or ***out***
- ***texture2D*** becomes ***texture***
- ***gl_FragColor*** no longer exist
- Etc.

ES 2.0 to ES 3.1: Shaders (cont.)

✦ Quick solution: preprocessor macro

Vertex shader

```
VTX_TO_FRG vec2 v_texcoord0;  // varying / out

ATTRIBUTE vec3 a_position;    // attribute / in
ATTRIBUTE vec2 a_texcoord0;   // attribute / in

void main()
{
  v_texcoord0 = a_texcoord0;
}
```

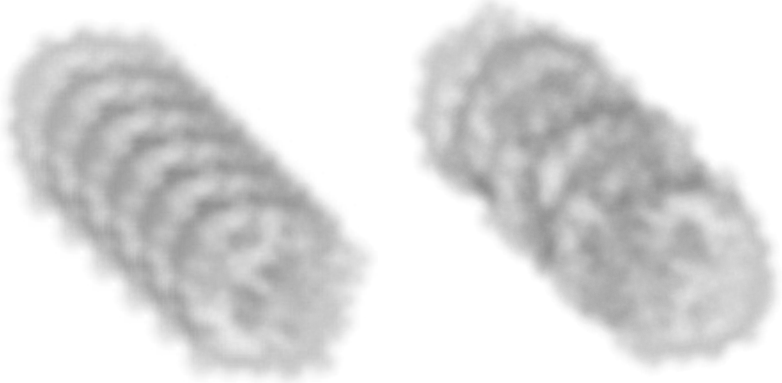
Fragment shader

```
VTX_TO_FRG vec2 v_texcoord0;  // varying / in

void main()
{
  v_texcoord0 = a_texcoord0;

  OUT_FRAGCOLOR = vec4(…);  // gl_FragColor / out vec4
}
```

Geometry shaders

- **GL_INTEL_geometry_shader** for **particles**
- CPU fallback otherwise
  - 4 verts, 2 triangles
- No **GL_POINTS**, because…
  - Limits on size
  - No rotation! *
  - Clips as a point on desktop
    (fixed in ES, still bad for multi-platform)

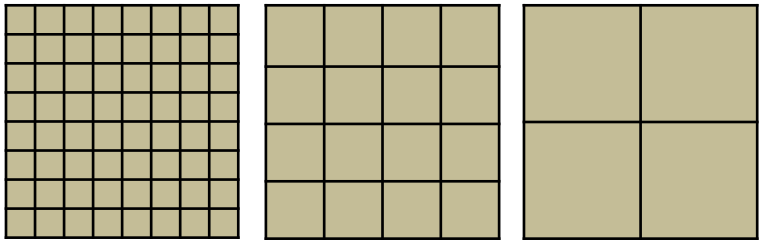
No Rotation **With Rotation**

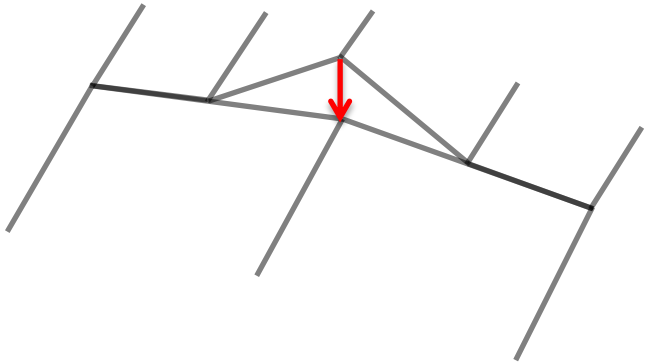


* Rotation is technically possible, but heavy on fill rate

Terrain without tessellation shaders

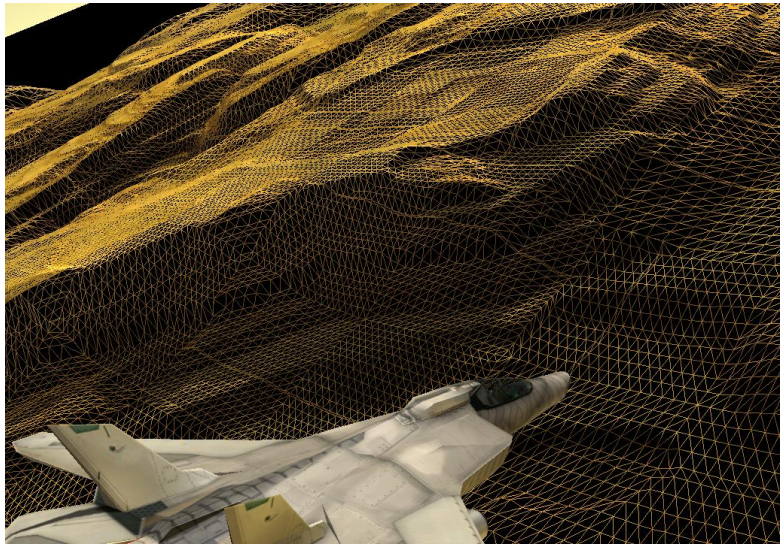
- Terrain is logically divided in patches
- Setup creates grids at different LODs (64x64, 32x32, …)
- For each frame
  - CPU decides viable LOD
- For each grid
  - CPU prepares height displacement for each vertex
    - Care is taken to close gaps between LODs
  - Draw call is issued





Terrain *with* tessellation shaders

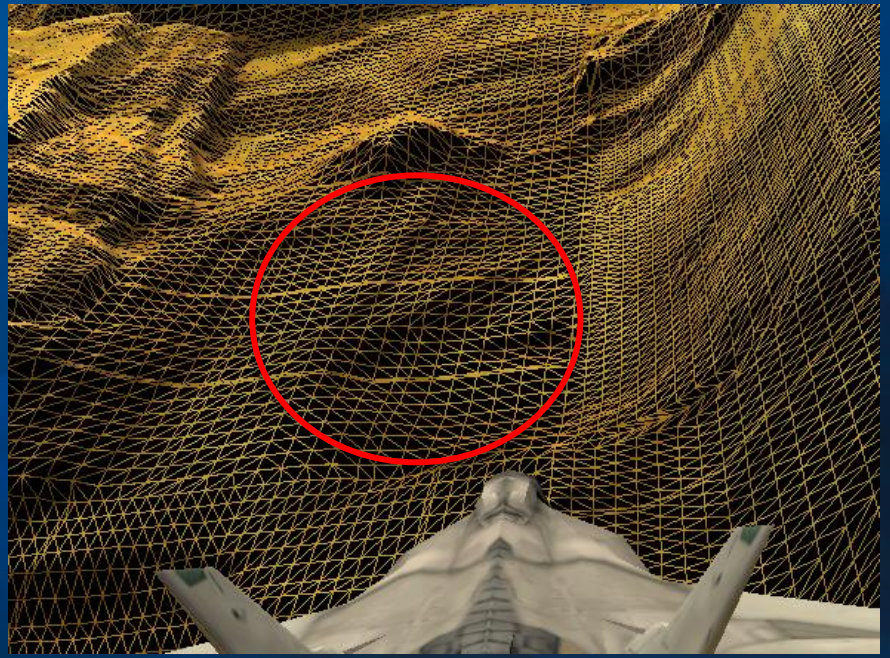
- When GL_INTEL_tessellation is available
  - Fixed grid meshes are replaced by GL_PATCHES
- For each frame
  - CPU picks LOD to pass to TCS
  - Don't want TSC to be "too smart".
    It can lead to funny wobbling effect
- For each patch (or grid)
  - CPU picks resolution (subdivisions) for each edge of the patch
    - This goes straight to gl_TessLevelOuter[0..3] in the Tess. Control Shader
  - Draw call is issued



Predefined vs variable LODs

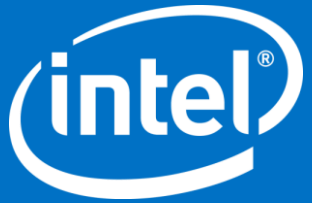
Sharp LOD jump (CPU)

Even LOD distribution (TS)



Terrain with tessellation: conclusions

- Still 1 draw call per patch (lazy implementation)
  - But more patches, increasing drawing distance
- CPU work is practically null. Just quick LOD estimation
- Finer subdivisions allows for better vertex count economy
- So much simpler and more efficient than DIY tessellation
  - Never again without!
- Don’t get too happy with dynamic subdivision
  - It can look “alive” if not nearly sub-pixel!
- Tessellation for other assets?
  - If you can sell it to the artists ☺



Geometry shaders (cont.)

- **Works just like on desktop OpenGL**
- **Get** gl_in[0].gl_Position **from vertex shader**
- **Output 4 positions and tex coordinates**

```
float co = cos( rotAng ), si = sin( rotAng );

vec3 pos_VS = gl_in[0].gl_Position.xyz;

gl_Position = makeRotPos( pos_VS,  hsiz, hsiz, co, si );
txCoord_GS = vec2( txCoord_VS[0]+txSiz[0], txCoord_VS[1] );
EmitVertex();

gl_Position = makeRotPos( pos_VS, -hsiz, hsiz, co, si );
txCoord_GS = vec2( txCoord_VS[0]         , txCoord_VS[1] );
EmitVertex();

//...

EmitPrimitive();
```