Shaders I

Idaho State University | Computer Science

Isaac Griffith

CS 4458 and CS 5558
Department of Informatics and Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outcomes

At the end of Today's Lecture you will be able to:

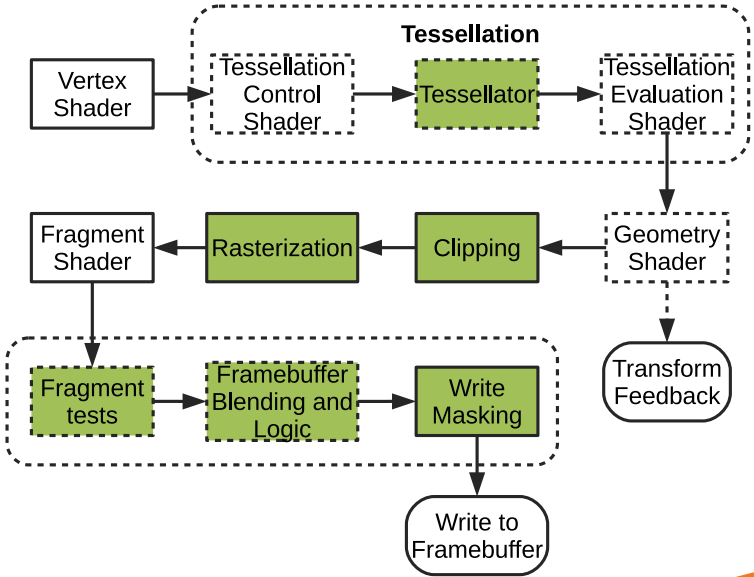
- Understand the basic concepts of the current OpenGL pipeline
- Begin to think in the streaming way of shaders

ROAR

Shader Programming: Basic Idea

- Replace vertex and/or fragment computations with user program, or “shader”
- Shaders are small, stateless programs run on the GPU with a high degree of parallelism
- Written in a high-level language that hides parallelism from the programmer: GLSL, HLSL, Cg
- Graphics driver compiles shader and links them into a program at application run-time, within an OpenGL or Direct3D program

The New Pipeline



Shaders

- Vertex Shader:
  - Mandatory shading stage
  - Runs once per input vertex
  - Must output one (x, y, z, w) coordinate for each vertex
- Tesselation Shader:
  - Optional shading stage
  - Comprised of two shaders:
    - Tessellation Control Shader
    - Tessellation Evaluation Shader
  - Adaptively refines or coarsens an object's mesh

Shaders

- Geometry Shader:
  - Optional shading stage
  - Allows transformation of original geometry
- Fragment Shader:
  - Mandatory shading stage
  - runs once per output Fragment
  - Must compute (at least) interpolated color values

The Vertex Shader

Vertex shader replaces

- vertex transformation
- normal transformation, normalization
- lighting
- texture coordinate generation and transformation

Vertex Shader

- Input: individual vertex in model coordinates
- Output: Individual vertex in clip (cvv) coordinates
- Operate on individual vertex
  - results generated from one vertex cannot be shared with other vertices
- Cannot create or destroy vertex
- Must do:
  - transforms
  - lighting

Idaho State University
Computer Science

Why Use Vertex Shader?

- Complete control of transform and lighting hardware
- Custom vertex lighting
- Custom vertex computations
  - custom skinning and blending
  - object/character deformation
  - procedural deformation
- Custom texture coordinate generation
- Custom texture matrix operations
- Complex vertex operations accelerated in hardware
- Offloading vertex computations frees up CPU
  - more physics and simulation possible!
  - particle systems

ROAR

Idaho State University
Computer Science

The Fragment/Pixel Shader

Fragment Shader replaces:

- texture accesses & application
- fog and some fragment tests

Has access to OpenGL states

ROAR

Fragment/Pixel Shader

- Input: individual fragment in window coordinates
- Output: individual fragment in window coordinates
- Operate on individual fragment
  - Results generated from one fragment cannot be shared with other framents
- Texture coordinates
  - Store information for lookup in textures
  - More general than images-to-be-glued, e.g., bump mapping

Why Fragment Shaders?

- Per-pixel lighting
  - looks much better than per-vertex lighting
    - true Phong shading
  - per-pixel Fresnel term and Cook-Torrance lighting
  - anistropic lighting
  - non-photorealistic rendering (NPR)
    - Toon shading, htaching, Gooch lighting, image space techniques
- Volumetric effects
- Advanced bump mapping
- Procedural textures and texture perturbation

Shader Programming Model

- The GPU is a stream processor
- Each point in 3D space has one or more attributes defining it:
  - position
  - surface normal
  - color
  - texture coordinates
  - etc.
- Each attribute forms a stream of data that are fed to the shaders one after another
- From the application to the vertex shader, each data stream contains the values of the same attribute (such as position) across vertices
- The vertex shader is applied to each vertex of each graphics primitive in a stateless manner.

Shader Programming Model

- The rasterizer:
  - does scan conversion: figures out which fragments are covered by the primitive, and
  - interpolates an attribute across vertices to computer the corresponding per-fragment values
- Data streams arriving at the fragment shader contain per-fragment attribute values (a much larger set than per-vertex attribute streams@)
- The fragment shader is applied to each fragment covered by a graphics primitive in a stateless manner.

Idaho State University
Computer Science

Shader Programming: Outline

- Basic GLSL Syntax
- Vertex Shader with example
- Fragment shader, with vertex+fragment shader example
- GLSL syntax for vectors and matrices
- GLSL built-in functions
- Data passing in GLSL
- Example vertex to grament shader data passing
- Built-in GLSL global variables
- Example: built-in global variables usage
- Integrating GLSL with OpenGL

ROAR

GLSL

- Comes with OpenGL
- Based on C, with some C++ features
- Restricted programming model, to allow for transparent parallelization, threading, and load balancing
- Graphics-friendly data types:

```
void, bool, int, float, // no double
vec2, vec3, vec4, // default to float
ivec2, bvec2, [ib]vec[34],
mat2, mat3, mat4, // square matrices only
structs, 1D arrays, functions
```

Control Flow

- C-like expression for execution control:
  - `if (bool) ... else ...`
  - `for (i = 0; i < n; i++) loop`
  - `do .. while (bool)`
- However, these conditional branching is much more expensive than in CPU $\rightarrow$ don't use too much of it, especially in fragment shader

THINK PARALLEL!

- Same code applies to all vertices/fragments

Idaho State University
Computer Science

Shader Code Snippet

```
void main() {
  const float f = 3.0;
  vec3 u(1.0), v(0.0, 1.0, 0.0);

  for (int i = 0; i < 10; i++)
    v = f * u + v;

  ...
}
```

- Seems like general purpose computing
  - What's missing?

ROAR

Missing Features

- No points, no dynamically allocated memory
- No recursion
- No strings, no characters
- No double, byte, short, long, ...
- No file I/O
- No `printf()`
- focus is on (parallel) numerical computation

Strong Typing

- No automatic type conversion

```
float f = 1; // WRONG
float f = 1.0; // much better
```

- Instead of casting, use constructors:

```
vec3 rgb = {1.0, 1.0, 1.0};
// c++ style constructor for type conversion:
vec4 rgba = vec4(rgb, 1.0);
vec2 rg = vec2(rgba);  // and masking
```

Example: Minimal Vertex Shader

What a vertex shader must minimially do (what the rasterizer expects):

- transform vertex position from model to eye coordinates
- and then project to clip (cvv) coordinates
- finally, output the vertex position
- Compute vertex lighting if `GL_LIGHTING` is on

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

- all `gl_*` variables above are part of the OpenGL state, which a shader can access without declaring

Idaho State University
Computer Science

Example: From Vertex Shader

```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 modelViewMat;
uniform mat4 projMat;

out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```


ROAR

Example: to the Rasterizer …

The rasterizer interpolates vertex attributes, such as positions and colors, across a primitive and generates the corresponding attributes for each fragment forming the primitive.

Example: to the Fragment Shaders

In the following fragment shader:

```
#version 430 core

in vec4 colorsExport;

out vec4 colorsOut;

void main(void)
{
    colorsOut = colorsExport;
}
```

Idaho State University
Computer Science

Vector Components

Vector components can be accessed by: * position $(xyzw)$, color $(rgba)$, texture-coordinates $(stpq)$ - these are syntactic sugar only - they can't be mixed in a single selection * or plain index: `a[i]`

```
vec2 v2;
vec3 v3;
vec4 v4;

v2.x    // return a float
v2.z    // wrong: undefined for type
v4.rgba // returns a vec4
v4.stp  // returns a vec3
v4.b    // returns a float
v4.xy   // returns a vec2
v4.xgp  // wrong: mismatched component sets
```


ROAR

Swizzling & Smearing

Swizzling operator lets you access any particular component(s) of a vector

- swizzle as R-values:

```
vec2 v2;
vec4 v4;
v4.wzyx // swizzles, returns a vec4
v4.bgra // swizzles, returns a vec4
v4.xxxx // smears x, returns a vec4
v4.xxx  // smears x, returns a vec3
v4.yyxx // swizzles and smears x and y,
        // returns a vec4
v2.yyyy // wrong: too many components for type
v2.xy = v2.yx // swaps components
```

Swizzling

- swizzle as L-values

```
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);

v4.xw = vec(5.0, 6.0); // (5.0, 2.0, 3.0, 6.0)
v4.wx = vec2(7.0, 8.0); // (8.0, 2.0, 3.0, 7.0)
v4.xx = vec(9.0, 10.0); // wrong: x used twice
v4.yz = 11.0;           // wrong: type mismatch
v4.yz = vec2(12.0);     // (8.0, 12.0, 12.0, 7.0)
```

Example: compute cross product $n = u \times v$

$$\begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

```
n = u.yzx * v.zxy - u.zxy * v.yzx;
```

Advantage: avoid intermediates and copies

Idaho State University
Computer Science

Matrix Components

Matrices are created, stored, and accessed in column major order * `M[i]`: column `i` of `M`, as in C/C++, starts from 0 * `M[i][j]`: element in col `i`, row `j`

The `*` operator in GLSL is overloaded, hence:

```
mat4 M;
vec4 u, v, w;

u = M * v;
w = v * M;
```

work and compute the right, **and different**, results for `u` and `w`

Matrix Constructors

```
vec2 v1, v2;
mat2 m2;
mat4 m4;

mat4(1.0, 2.0, 3.0, 4.0,   // first column
     5.0, 6.0, 7.0, 8.0,   // second column
     9.0, 10., 11., 12.,   // third column
     13., 14., 15., 16.);  // fourth column

v1 = vec2(1.0, 2.0);
v2 = vec2(3.0, 4.0);
m2 = mat2(v1, v2);         // 1st col: 1.0, 2.0,
                           // 2nd col: 3.0, 4.0
mat4(1.0);                 // identity matrix
mat3(m4);                  // upper 3x3
vec4(m4);                  // first column
float(m4);                 // upper 1x1
```

Build-in Functions

Common:

- `abs, float, ceil, min, max, mod(dividend, divisor), sign(x) // 1.0 if x > 1, 0.0 if x = 0, else -1.0 fract(x) // x - floor(x) clamp(x, low, high) // min(max(x, low), high)`

Exponentials:

- `pow, exp2, log2, sqrt, inversesqrt (1/sqrt)`

Angles & trigonometry:

- `radians, degrees, sin, cos, tan, asign, acos, atan`

Interpolations:

- `mix(x, y, a)` `// (1.0 - a)x + ay, D3D: lerp(x, y, a)`
- `step(edge, x)` `// x < edge ? 0.0 : 1.0`
- `smoothstep(edge0, edge1, x)` `// Hermite interpolation`

Built-in Functions

Geometric:

- `length (of vector), distance (between 2 points), cross, dot, normalize, faceForward, reflect (about normal)`

Matrix:

- `matrixCompMult`

Vector relational:

- `lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all`

Texture:

- `texture1D, texture2D, texture3D, textureCube`
- `texture1DProj, texture2DProj, texture3DProj, textureCubeProj`
- `shadow1D, shadow2D, shadow1DProj, shadow2Dproj`

Call-by Value-Return

Function parameters:

- **in**: copy in [default]
- **out**: copy out, undefined upon entrance
- **inout**: copy both

Note: no pointers or references

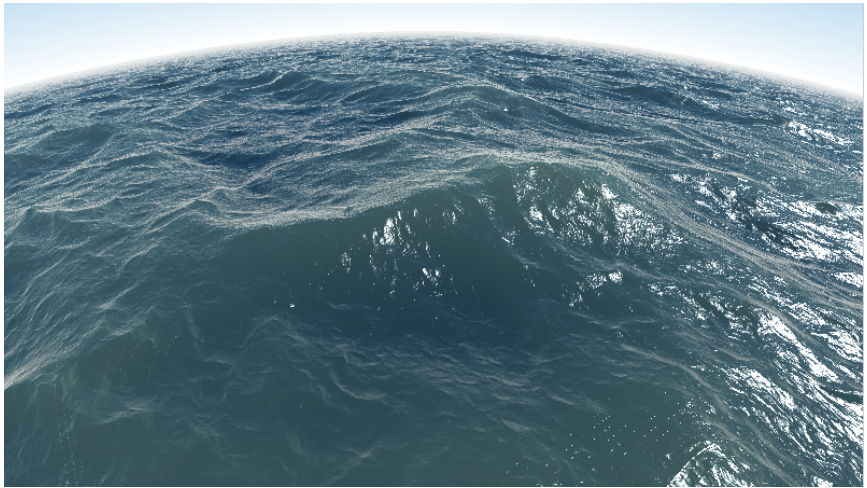
Data Passing w/ Global Vars

- OpenGL application passes data to the vertex shader using tow types of GLSL global variables: **uniform** and **attribute**
- The vertex shader passes data to the fragment shader by having the rasterizer interpolate **in** (in vs) values between vertices into per-fragment **out** (in vs), **in** (in fs) variables
- The fragment shader can also access **uniform** global variables set by the OpenGL application

Let’s look at an example!

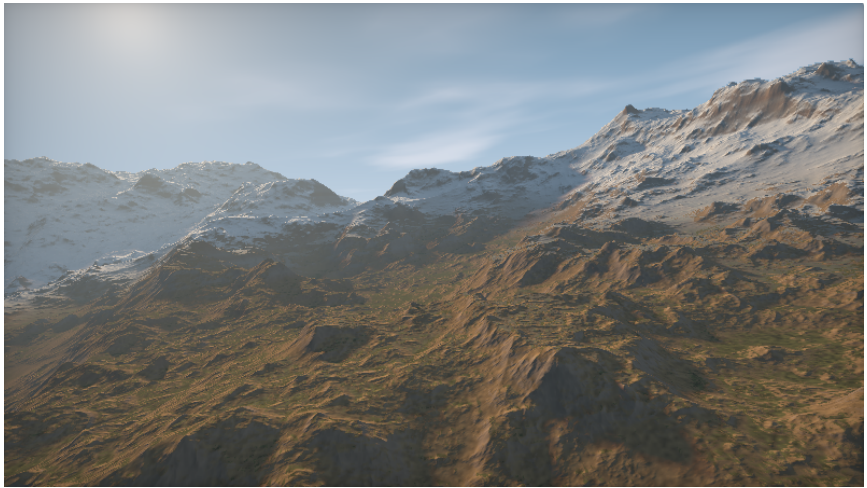
Examples

Seascape



Source Code

Elevated



Source Code

Structured Volume Sampling

Source Code

Flame

Source Code

Canyon

Source Code

Dolphin

Source Code

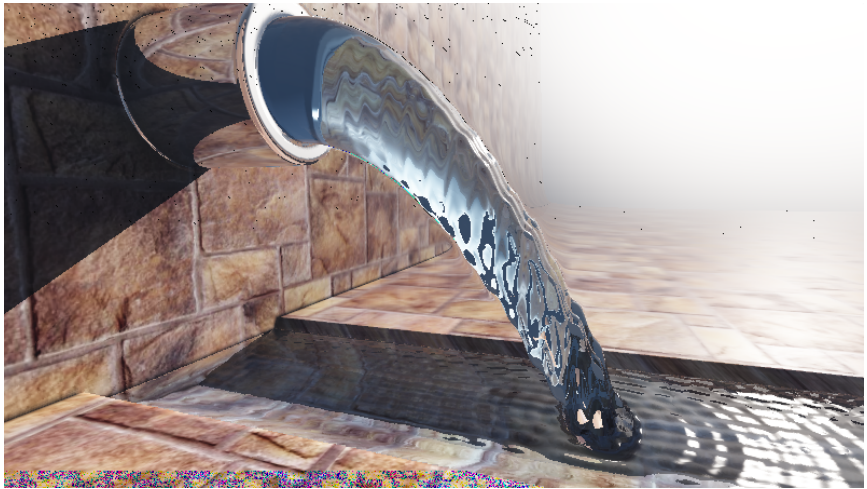
Fish Swimming

Source Code

Procedural Grass

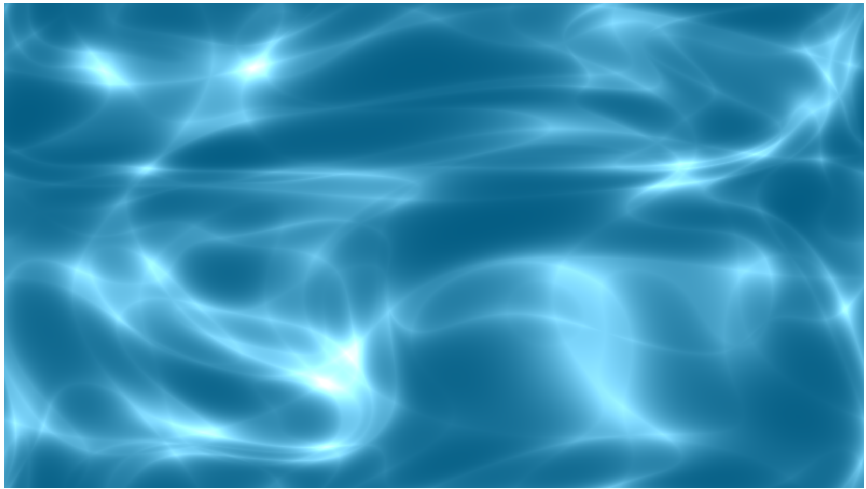
Source Code

Spout



Source Code

Tileable Water Caustic



Source Code

Smoke Columns

Source Code

Are there any questions?

Idaho State University
Computer Science

For Next Time

ROAR