CSCI 420 Computer Graphics
Lecture 7

Shaders

Shading Languages
GLSL
Vertex Array Objects
Vertex Shader
Fragment Shader
[Angel Ch. 1, 2, A]

Oded Stein
University of Southern California

Introduction

- The major advance in real time graphics has been the *programmable* pipeline:
  - First introduced by NVIDIA GeForce 3 (in 2001)
  - Supported by all modern high-end commodity cards
    - NVIDIA, AMD, Intel
  - Software Support
    - Direct3D
    - OpenGL

- This lecture: programmable pipeline and shaders

OpenGL Extensions

- Initial OpenGL version was 1.0
- Current OpenGL version is 4.6

- As graphics hardware improved, new capabilities were added to OpenGL
  - multitexturing
  - multisampling
  - non-power-of-two textures
  - shaders
  - and many more

OpenGL Grows via Extensions

- Phase 1: vendor-specific: GL_NV_multisample
- Phase 2: multi-vendor: GL_EXT_multisample
- Phase 3: approved by OpenGL’s review board GL_ARB_multisample
- Phase 4: incorporated into OpenGL (v1.3)

OpenGL 2.0 Added Shaders

- Shaders are customized programs that replace a part of the OpenGL pipeline

- They enable many effects not possible by the fixed OpenGL pipeline

- Motivated by Pixar's Renderman (offline shader)

Shaders Enable Many New Effects

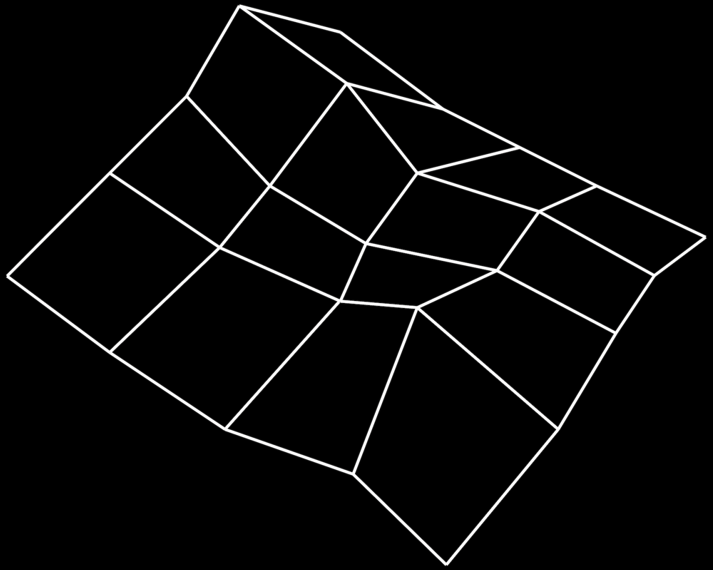
Complex materials

Shadows

Lighting environments

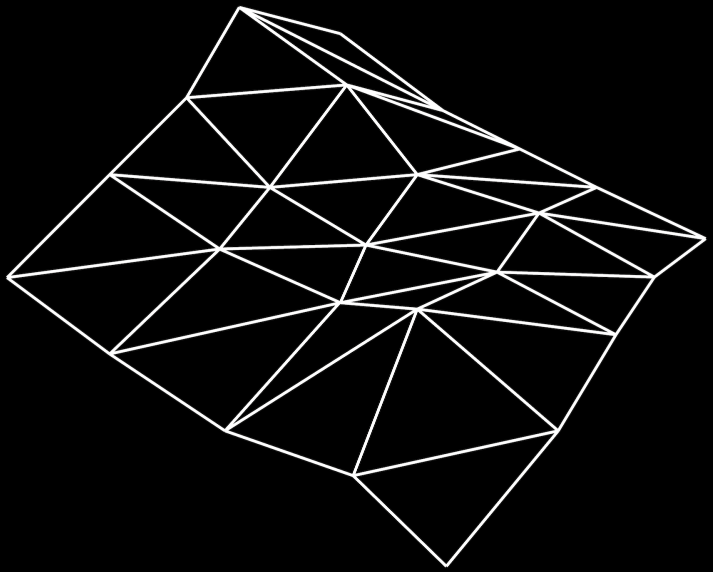
Advanced mapping

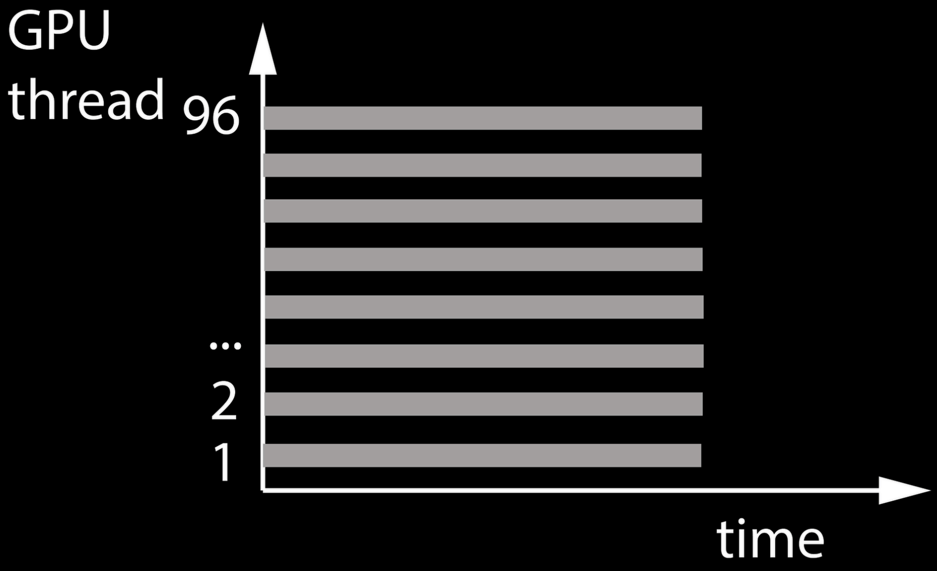
Vertex Shader



5x5 terrain (as in hw1)
5x5 = 25 vertices
4x4 = 16 quads

Vertex Shader

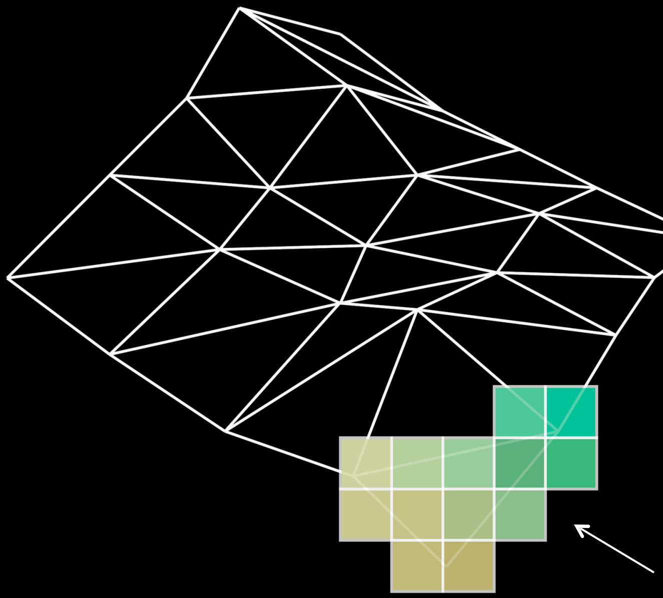




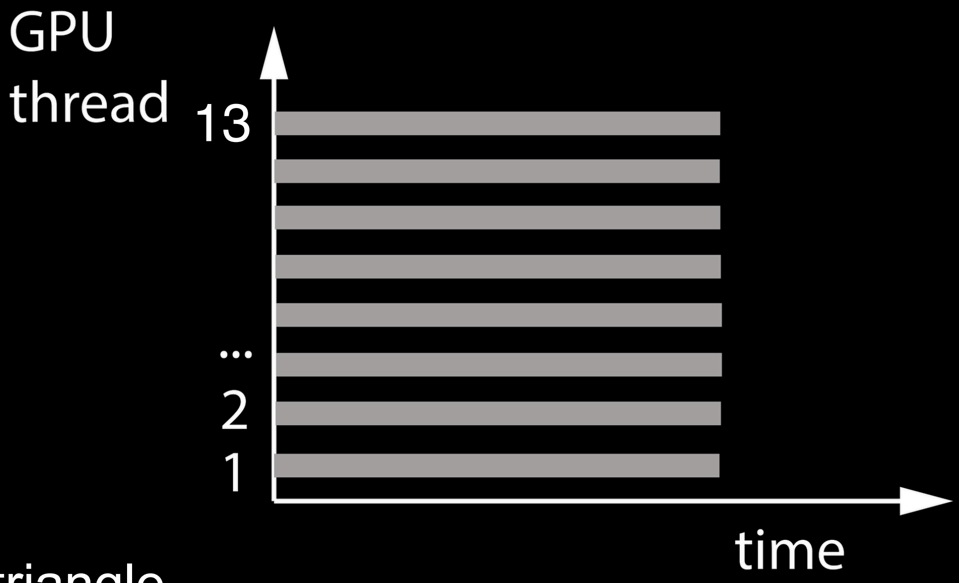
96 vertex shaders execute in parallel

User must tessellate into triangles (in the VBO)
4 x 4 x 2 = 32 triangles
32 x 3 = 96 vertices (assuming GL_TRIANGLES)

Fragment Shader



This triangle rasterizes into 13 pixels

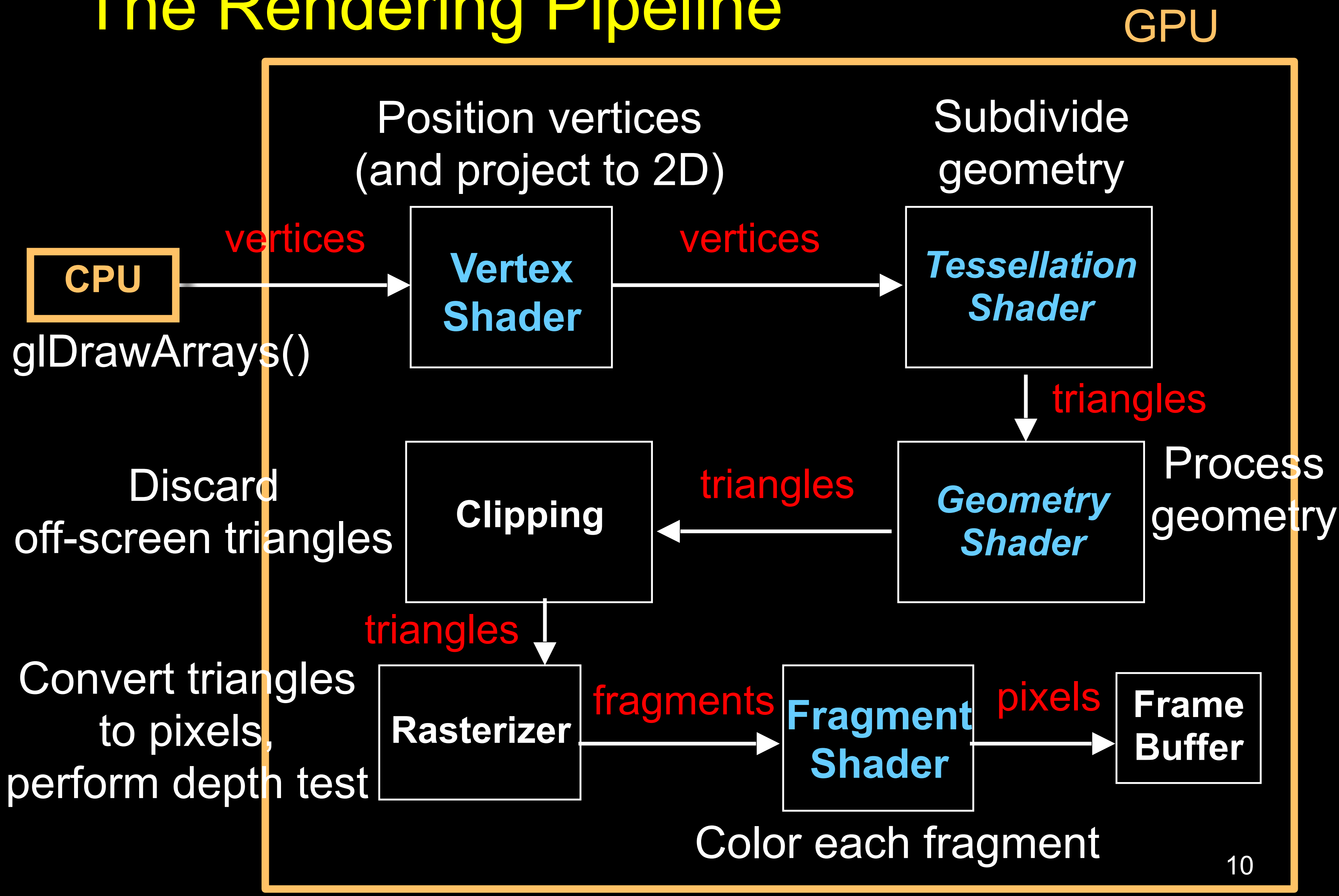


13 fragment shaders execute in parallel

Rasterization

Some pixels may repeat in multiple triangles

The Rendering Pipeline



GPU

CPU

glDrawArrays()

vertices

Position vertices
(and project to 2D)

Vertex Shader

vertices

Subdivide
geometry

Tessellation Shader

triangles

Process
geometry

Geometry Shader

triangles

Discard
off-screen triangles

Clipping

triangles

Convert triangles
to pixels,
perform depth test

Rasterizer

fragments

Fragment Shader

Color each fragment

pixels

Frame Buffer

Shaders

- Vertex shader (= vertex program)

- Tessellation control and evaluation shader
  (OpenGL 4.0; subdivide the geometry)

- Geometry shader
  (OpenGL 3.2; process, generate, replace or delete geometry)

- Fragment shader (= fragment program)

- Compute shader (OpenGL 4.3; general purpose)

Shaders

- Compatibility profile: Default shaders are provided by OpenGL *(fixed-function pipeline)*
- Core profile: no default vertex or fragment shader; must be provided by the programmer
- Tessellation shaders, geometry shaders and compute shaders are *optional*

Shader Variables Classification

- Attribute
  - Information specific to each vertex/pixel passed to vertex/fragment shader

Example:
Vertex Color

- Uniform
  - Constant information passed to vertex/fragment shader
  - Cannot be written to in a shader

Example:
Light Position
Eye Position

- Out/in
  - Info passed from vertex shader to fragment shader
  - Interpolated from vertices to pixels
  - Write in vertex shader, but only read in fragment shader

Example:
Vertex Color
Texture Coords

- Const
  - To declare non-writable, constant variables

Example:
pi, e, 0.480

Shaders Are Written in *Shading Languages*

- Early shaders: assembly language
- Since ~2004: high-level shading languages
  - OpenGL Shading Language (GLSL)
    - highly integrated with OpenGL
  - Cg (NVIDIA and Microsoft), very similar to GLSL
  - HLSL (Microsoft), the shading language of Direct3D
  - All of these are simplified versions of C/C++

GLSL

- The shading language of OpenGL
- Managed by OpenGL Architecture Review Board
- Introduced in OpenGL 2.0
- We use shader version 1.50:
  #version 150
  (a good version supporting the core profile features)
- Current shader version: 4.60
  - (it's been that for a while - 2018)

Vertex Shader

- Input: vertices, in object coordinates, and per-vertex attributes:
  - color
  - normal
  - texture/UV coordinates
  - many more
- Output:
  - vertex location in clip coordinates
  - vertex color
  - vertex normal
  - many more are possible

Basic Vertex Shader in GLSL

```
#version 150
in vec3 position; // input position, in object coordinates
in vec4 color; // input color
out vec4 col; // output color

uniform mat4 modelViewMatrix; // uniform variable to store the modelview mtx
uniform mat4 projectionMatrix; // uniform variable to store the projection mtx

void main()
{
  // compute the transformed and projected vertex position (into gl_Position)
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0f);
  // compute the vertex color (into col)
  col = color;
}
```

Fragment Shader

- Input: fragments (tentative pixels), and per-pixel attributes:
  - color
  - normal
  - texture coordinates
  - many more are possible

- Inputs are outputs from the vertex shader, interpolated (by the GPU) to the pixel location !

- Output:
  - pixel color
  - depth value
  - can discard the fragment using the **discard** keyword

Basic Fragment Shader

```
#version 150

in vec4 col; // input color (computed by the interpolator)
out vec4 c; // output color (the final fragment color)

void main()
{
  // compute the final fragment color
  c = col;
}
```

Another Fragment Shader

```
#version 150

in vec4 col; // input color (computed by the interpolator)
out vec4 c; // output color (the final fragment color)

void main()
{
  // compute the final fragment color
  c = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Perspective projection

- Where do we handle the final perspective divide?

- Is it even possible in a shader? Or should OpenGL handle it for us?

- Exercise!

Pipeline program

- Container for all the shaders
- Vertex, fragment, geometry, tessellation, compute
- Can have several pipeline programs
  (for example, one for each rendering style)
- Must have at least one (core profile)
- At any moment of time, exactly one pipeline program is bound (active)

Installing Pipeline Programs

- Step 1: Create Shaders
  - Create handles to shaders
- Step 2: Specify Shaders
  - load strings that contain shader source
- Step 3: Compiling Shaders
  - Actually compile source (check for errors)
- Step 4: Creating Program Objects
  - Program object controls the shaders
- Step 5: Attach Shaders to Programs
  - Attach shaders to program objects via handle
- Step 6: Link Shaders to Programs
  - Another step similar to attach
- Step 7: Enable Shaders
  - Finally, let OpenGL and GPU know that shaders are ready

Our helper library: PipelineProgram

```
// load shaders from a file
int BuildShadersFromFiles(const char * filenameBasePath,
       const char * vertexShaderFilename,
       const char * fragmentShaderFilename,
       const char * geometryShaderFilename = NULL,
       const char * tessellationControlShaderFilename = NULL,
       const char * tessellationEvaluationShaderFilename = NULL);
```

Our helper library: PipelineProgram

```
// load shaders from a C text string
int BuildShadersFromStrings(const char * vertexShaderCode,
       const char * fragmentShaderCode,
       const char * geometryShaderCode = NULL,
       const char * tessellationControlShaderCode = NULL,
       const char * tessellationEvaluationShaderCode = NULL);
```

Setting up the Pipeline Program

```
// global variable
PipelineProgram pipelineProgram;

// during initialization:
pipelineProgram.BuildShadersFromFiles(“../openGLHelper”,
   "vertexShader.glsl", "fragmentShader.glsl”);

// before rendering, bind (activate) the pipeline program:
pipelineProgram.Bind();
```

If you want to use a different pipeline program,
then “Bind” that other pipeline program.

Setting up the Uniform Variables

Uploading the modelview matrix transformation to the GPU
(in the display function)

```
float m[16]; // column-major
// here, must fill m (missing code; use OpenGLMatrix class)
 // …

// upload m to the GPU
 pipelineProgram.Bind();
GLboolean isRowMajor = GL_FALSE;
 pipelineProgram->SetUniformVariableMatrix4fv(
    "modelViewMatrix", isRowMajor, m);
```

Setting up the Uniform Variables

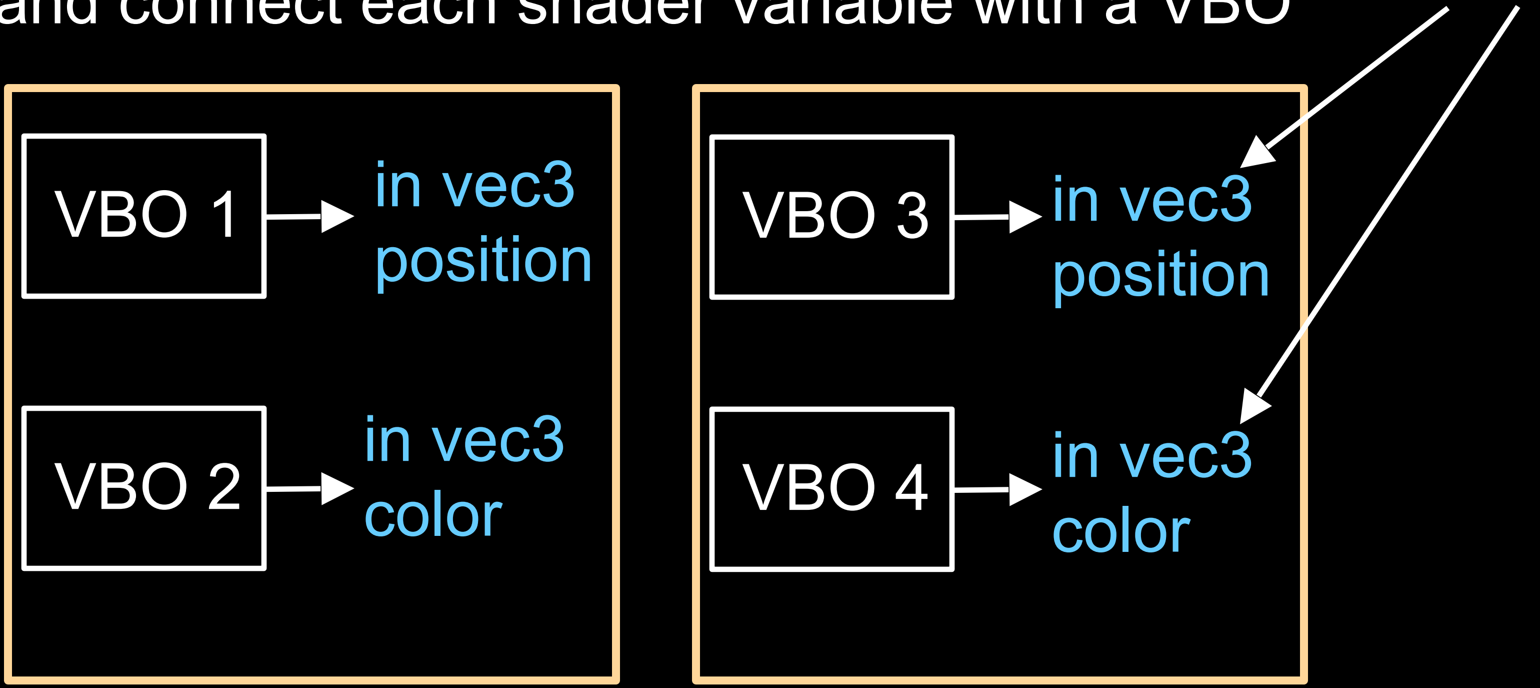
Repeat the same process also for the projection matrix:

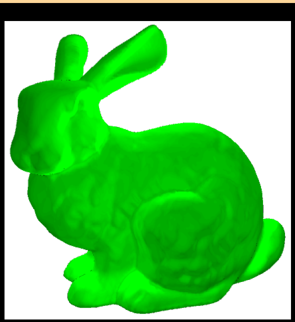
```
float p[16]; // column-major
// here, must fill p (missing code; use OpenGLMatrix class)
 // …

// upload p to the GPU
 pipelineProgram.Bind();
GLboolean isRowMajor = GL_FALSE;
 pipelineProgram->SetUniformVariableMatrix4fv(
     ”projectionMatrix", isRowMajor, p);
```

Vertex Array Objects (VAOs)

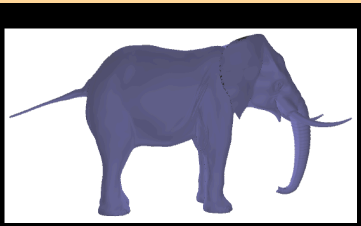
- A container to collect the VBOs of each object and connect each shader variable with a VBO

shader variables





VAO 1



VAO 2

Vertex Array Objects (VAOs)

- A container to collect the VBOs of each object
- Usage is mandatory (by the OpenGL standard)
- During initialization:
  - create VBOs (one or more per object),
  - create VAOs (one per object),
  - place the VBOs into the VAO, and connect VBOs to shader variables
- At render time: bind the VAO, then call glDrawArrays()

VAO code (initialization)

During initialization:

```
// Create a VAO.
VAO * vao = new VAO();
// Connect the shader variables to their respective VBOs.
vao->ConnectPipelineProgramAndVBOAndShaderVariable(
    pipelineProgram, vboPositions, “position”);
vao->ConnectPipelineProgramAndVBOAndShaderVariable(
    pipelineProgram, vboColors, “color”);
```

Using the VAO for rendering

In the display function:

```
// Bind the vertex and fragment shaders to use.
pipelineProgram->Bind();
// Select which object to render.
vao->Bind();

// Render the object contained in the VAO.
GLint first = 0;
GLsizei count = numVertices;
glDrawArrays(GL_TRIANGLES, first, count);
```

GLSL: Standard

- GLSL is a well-specified standard
  - https://www.khronos.org/opengles/sdk/docs/manglsl/docbook4/
  - Really useful online reference
- Doesn't always work as specified... depends on the driver vendor.
- Don't write crazy C code using syntax features.
  - But do use crazy math!

GLSL: Data Types

- Scalar Types
  - float - 32 bit, very nearly IEEE-754 compatible
  - int - at least 16 bit
  - bool - like in C++
- Vector Types
  - vec[2 | 3 | 4] - floating-point vector
  - ivec[2 | 3 | 4] - integer vector
  - bvec[2 | 3 | 4] - boolean vector
- Matrix Types
  - mat[2 | 3 | 4] - for 2x2, 3x3, and 4x4 floating-point matrices
- Sampler Types
  - sampler[1 | 2 | 3]D - to access texture images

GLSL: Data Types

- Structs
  - Very useful to shuffle specific information between CPU and GPU in one block.
  - Careful - padding!

```
struct Light {
      float intensity;
      vec3 position;
      vec4 color;
};

.....

uniform Light l1;

.....

void main()
{
      Light l = ...;
}
```

GLSL: Operations

- Operators behave like in C++
- Component-wise for vector & matrix
- Multiplication on vectors and matrices

- Examples:
  - `Vec3 t = u * v;`
  - `float f = v[2];`
  - `v.x = u.x + f;`

GLSL: Swizzling

- Swizzling is a convenient way to access individual vector components

```
vec4 myVector;
myVector.rgba; // is the same as myVector
myVector.xy; // is a vec2
myVector.b; // is a float
myVector[2]; // is the same as myVector.b
myVector.xb; // illegal
myVector.xxx; // is a vec3

0123 ~ xyzw
```

GLSL: Flow Control

- Loops
  - C++ style if-else
  - C++ style for, while, and do

Example:

```
if(isHidden) {
    color.w = 0.;
}

...

for(int i=0; i<numLights; ++i)
{
   color += light[i].c;
}
```

GLSL: Flow Control

- Loops
  - C++ style if-else
  - C++ style for, while, and do

- Jumps
  - `continue;`
  - `break;`
  - `return;`
  - `discard;` (in fragment shader)

GLSL: Flow Control

- Functions
  - Much like C++
  - Entry point into a shader is void main()
  - No support for recursion
  - Call by value-return calling convention

Example:

```
float sign(float x) {
    if(x>0) {
        return 1.;
    } else if(x<0) {
        return -1.;
    } else {
        return 0.;
    }
}
```

GLSL: Flow Control

- Functions
  - C++-like overloading exists.
  - Arguments can be `const`
  - You can add a parameter qualifier
    - `in` (default; will be copied into variable on function call)
    - `out` (will be copied into variable on function return)
    - `inout` (will be copied into variable on function call and return)
  - `const` contradicts `out` and `inout`
  - Poor man's pass-by-reference
    - Not actual references, only copying.
    - Avoid if you can because of confusion.

GLSL: Flow Control

Example:

```
void sign(in float x, out float y) {
    if(x>0) {
        y = 1.;
    } else if(x<0) {
        y = -1.;
    } else {
        y = 0.;
    }
}
```

GLSL: Built-in Functions

- Wide Assortment
  - Trigonometry (cos, sin, tan, etc.)
  - Exponential (pow, log, sqrt, etc.)
  - Common (abs, floor, min, clamp, etc.)
  - Geometry (length, dot, normalize, reflect, etc.)
  - Relational (lessThan, equal, etc.)

- Need to watch out for common reserved keywords
- Always use built-in functions, do not implement your own
  - Sign function from last slide is a bad idea.
- Some functions are not implemented on some cards

GLSL: Built-in Variables

- Always prefaced with gl_
- Accessible to both vertex and fragment shaders
- Examples:
  - (input) gl_VertexID: index of currently processed vertex
  - (input) gl_FrontFacing: whether pixel is front facing or not
  - (input) gl_FragCoord : x,y: coordinate of pixel, z: depth
  - (output) gl_FragDepth: pixel depth

Debugging Shaders

- More difficult than debugging C programs

- Common show-stoppers:
  - Typos in shader source
  - Assuming implicit type conversion (cannot convert vec4 to vec3)
  - Attempting to connect VAOs to non-existent (say, due to a typo) shader variables
  - Driver bugs beyond your control
    - https://stackoverflow.com/a/23999304

- Very important to check error codes; use status functions like:
  - `glGetShaderiv(GLuint shader, GLenum pname, GLint * params)`

A few debugging best practices

- printf debugging corresponds to just writing into an output color

```
if(condition) {
  col = vec4(1., 0., 0., 1.);
} else {
  col = vec4(0., 1., 0., 1.);
}
```

- Pass variables through to fragment shader.
- Display input variables.
- Write multiple functions and color-debug them separately.

Summary

- Shading Languages
- Program Pipeline
- Vertex Array Objects
- GLSL
- Vertex Shader
- Fragment Shader

Shadertoy!

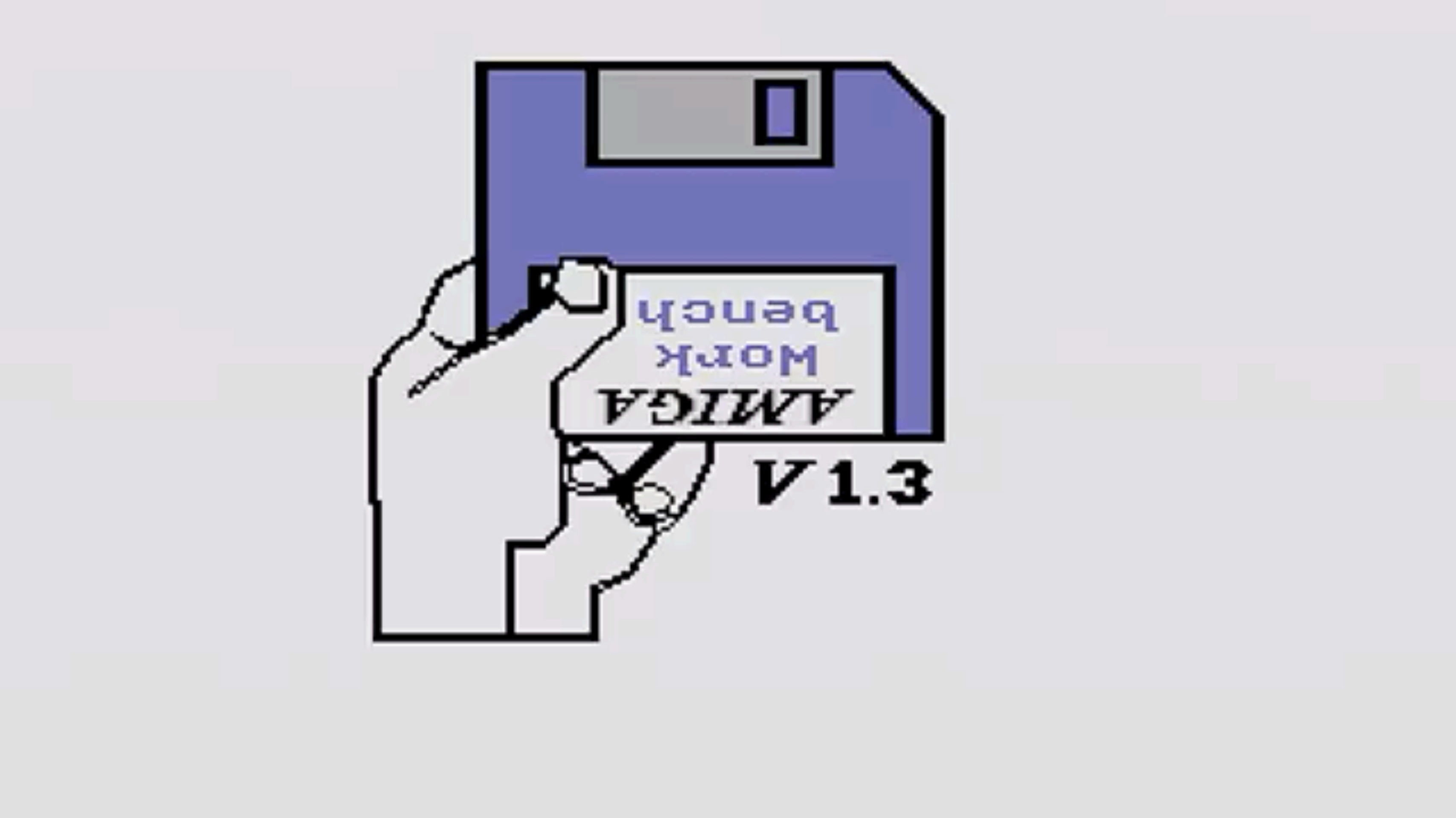
- www.shadertoy.com
- Website that allows you to write a fragment shader and run it in the web
- Fragment shaders are very powerful. You can do a lot with them

Demoscene

- Computer graphics subculture
- Writing small programs that show impressive visual videos
- File size restriction (1MB, 64KB, ...), old hardware...
- You can do a lot with just generating images with math, you don't need to save/load assets!
- Beautiful art form

Demoscene

- Running on 1MB memory Amiga
- Lotus https://www.pouet.net/prod.php?which=81094



Shadertoy!

- Basic example:
  - https://www.shadertoy.com/view/MlySzw
- More complicated examples:
  - https://www.shadertoy.com/view/MsjXDm
  - https://www.shadertoy.com/view/mtyGWy
- Crazy cool examples (too complicated for me to understand...):
  - https://www.shadertoy.com/view/4td3zj
  - https://www.shadertoy.com/view/MfjyWK
  - https://www.shadertoy.com/view/lsXGzH
  - https://www.shadertoy.com/view/lsf3zr