CS4621/5621 Fall 2015

Computer Graphics Practicum
Intro to OpenGL/GLSL
Part II

Professor: Kavita Bala
Instructor: Nicolas Savva

with slides from Balazs Kovacs, Eston Schweickart, Daniel Schroeder, Jiang Huang and Pramook Khungurn over the years.
Announcements

• Course website:
  http://www.cs.cornell.edu/courses/CS4621/2015fa/

• Piazza for practicum:
  https://piazza.com/cornell/fall2015/cs46215621/

• OpenGL Diagnostic v2

• First assignment GPUray out today (due Oct 15)
Today

- GPUr\text{ay}
- GLSL basics review
- OpenGL / LWJGL basics
Review
Pipeline

you are here → APPLICATION

3D transformations; shading → VERTEX PROCESSING

conversion of primitives to pixels → RASTERIZATION

blending, compositing, shading → FRAGMENT PROCESSING

user sees this → DISPLAY
GLSL passing data around

- Application
  - Triangles
  - Attributes
  - Vertex program
    - Varying parameters
  - Rasterizer
    - Varying parameters
    - Fragment program
      - Depth
      - Color
      - Framebuffer

- Uniform variables
GPUray Overview

- PPA1: port a simplified version of ray1 to GPU
- Mostly deal with the fragment shader
- LWJGL bindings (communicating with the GPU)
- Shader compiling and linking
- GLSL code
GPUray Demo
GPUray Tasks

- Setting up the CS4620/4621 framework and OpenGL
- Parsing ray1 xml scenes
- Writing GLSL shader code
- Creating the GLProgram and binding buffers to the GPU
- Adding user interaction
- Performance Benchmark
Creating an OpenGL Program using LWJGL revisited
MainGame

- A window which can display GameScreens
- Initializes OpenGL context
- Forwards keyboard and mouse events to the event dispatcher

Usage
  - Inherit from MainGame and implement methods
  - Create instance and call run method
GameScreen

• Can display images created by OpenGL
• OpenGL “context”
  – Stores OpenGL state (geometry, buffers, etc.)

• Usage:
  – Inherit from class and implement methods
  – Create instance in MainGame.buildScreenList
GameScreen

```java
@Override
public void update(GameTime gameTime) {
    // Animation: Update position of scene objects, camera
}

@Override
public void draw(GameTime gameTime) {
    // Drawing: Use LWJGL to draw to the screen
}

@Override
public void onEntry(GameTime gameTime) {
    // Initialization code
}

@Override
public void onExit(GameTime gameTime) {
    // Destruction, free allocated resources here
}
```
OpenGL Commands and Resources
OpenGL Commands

• Uses OpenGL context that is already initialized
• API calls: glxx.glSomeCommandName
• GL11.glClearColor(0.0f, 0.0f, 0.0f, 1.0f)
  - Set black as the color to use when clearing the screen
• GL11.glClear(GL11.GL_COLOR_BUFFER_BIT)
  - Clear the display buffer using the color given by glClearColor
• GL11.glDrawElements(…)
  - Draw primitives (now triangles)
Command Naming

• In C,
  • commands = functions
  • No two functions can have the same name.
  • Some commands take different arguments but do the same thing.

• All commands of the form
gl <name> {1234} {b s i f d ub us ui} {v}
Argument Types in Command Names

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Data Type</th>
<th>Typical Corresponding C-Language Type</th>
<th>OpenGL Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8-bit integer</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>s</td>
<td>16-bit integer</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>long</td>
<td>GLint, GLsizei</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point</td>
<td>float</td>
<td>GLfloat, GLclampf</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point</td>
<td>double</td>
<td>GLdouble, GLclampd</td>
</tr>
<tr>
<td>ub</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
<td>GLubyte, GLboolean</td>
</tr>
<tr>
<td>us</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>ui</td>
<td>32-bit unsigned integer</td>
<td>unsigned long</td>
<td>GLuint, GLenum, GLbitfield</td>
</tr>
</tbody>
</table>
Demo: Hello World!
Example: Hello World's draw()

```java
@override
public void draw(GameTime gameTime) {
    GL11.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    GL11.glClear(GL11.GL_COLOR_BUFFER_BIT);

    program.use();

    GLUniform.setST(program.getUniform("VP"),
                    new Matrix4(), false);
    GLUniform.set(program.getUniform("uGridColor"),
                  new Vector4(1, 1, 1, 1));

    vb.useAsAttrib(program.getAttribute("vPos"));
    ib.bind();
    GL11.glDrawElements(GL11.GL_TRIANGLES, indexCount,
                        GLType.UnsignedInt, 0);
    ib.unbind();

    GLProgram.unuse();
}
```
Framework Commands

• `program.use()`
  - Set which shader program the pipeline will use to draw geometry

• `GLUniform.setST(program.getUniform("VP"), …)`
  - Tell shader program to use the specified mat transformation as “VP”

• `GLUniform.set(program.getUniform("uGridColor"), …)`
  - Tell shader program to use the specified color as “uGridColor”

• `GLProgram.unuse()`
  - Tell OpenGL we are done drawing for now

• Each of these has OpenGL commands under the hood
Framework Commands

- `vb.useAsAttrib(program.getAttribute("vPos"))`
  - Tell shader program to use “vb” as vertex buffer and access vertex position using “vPos”

- `ib.bind(), ib.unbind()`
  - Bind (and unbind) the index buffer to tell OpenGL that we want to use the vertices in the vertex buffer in a certain way
Why Have a Framework?

• You write:
  
  vb.useAsAttrib(program.getAttribute("vPos"));

• Framework does:

  GL15.glBindBuffer(GL11.GL_ARRAY_BUFFER, vb.id);
  GL15.glEnableVertexAttribArray(program.getAttribute("vPos"));
  GL15.glVertexAttribPointer(program.getAttribute("vPos"), componentCount, componentFormat, norm, elementByteSize, offset * elementByteSize);

• Annoying to retype full sequence of commands for every draw
Framework and GL Resources

• OpenGL API has “objects” that hold rendering resources
  – Geometry, textures, shader programs, etc.
• Framework represents these with Java classes
  – GLProgram (shader programs)
  – GLBuffer (used to specify geometry)
• Constructing an object creates OpenGL resource
  – Object's data lives in GPU memory
  – Allows faster access while rendering
GLSL Program

• Specifies how OpenGL should draw geometry

• Program: A collection of shaders that run together
  - At least one vertex shader or one fragment shader

• At any time, the GPU runs only one program
  - Must specify program to use before drawing geometry
To use a GLSL program…

Follow the next 7 steps:

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shader.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use your shader program.
Now, to create a GLSL program...

- Create a GLProgram object

```java
private GLProgram program;

public void onEntry(GameTime gameTime) {
    program = new GLProgram();
    program.quickCreateResource("cs4621/demos/shaders/Grid.vert", // Path to vertex shader
                                "cs4621/demos/shaders/Grid.frag", // Path to fragment shader
                                null);                           // Optional attribute list
}
```
CS 4620/4621 Framework

• Contains GLProgram class to abstract OpenGL calls:
  – Added convenience methods
  – Help keep conventions straight
  – Controls mapping between attribute variables and vertex buffers
Back to GLSL
OpenGL/GLSL Plumbing

- Suppose we have already created the program.
- We tell OpenGL to use it.
- We then instruct OpenGL to draw the two triangles:

```java
public class HelloWorldScreen {
    // The vertices in our vertex buffer, initialized earlier
    float [] vertexPositions = {
        -0.5f, -0.5f,      // vertex 0
        0.5f, -0.5f,       // vertex 1
        0.5f,  0.5f,       // vertex 2
        -0.5f,  0.5f       // vertex 3
    };

    // In the draw method
    program.use();

    glDrawElements(...);

    GLProgram.unuse();
}
```
Modern Graphics Pipeline

- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T: Texture binding
- B: Buffer binding
Vertex Shader

- Transform vertices from object space to clip space

- Compute other data that are interpolated with vertices
  - Color
  - Normals
  - Texture coordinates
  - Etc.
Fragment Shaders

- Compute the color of a fragment (i.e. a pixel)
- Take interpolated data from vertex shaders
- Can read more data from:
  - Textures
  - User specified values
Green Triangle Demo
**green_shader.vert:**

```cpp
#version 120

uniform mat4 VP;
attribute vec3 vPos;

void main()
{
    gl_Position = VP * vec4(vPos,1);
}
```
green_shader.frag:

```glsl
#version 120

void main()
{
    gl_FragColor = vec4(0, 1, 0, 1);
}
```

Fragment Shader
GLSL Data Types

• Both in GLSL and Java
  • float, int
• GLSL has, but Java does not have
  • vec2, vec3, vec4: vectors
  • mat2, mat3, mat4: matrices
  • sampler1D, sampler2D, sample3D, samplerCube, etc: textures
• Java has, but GLSL does not have
  • Object
  • String
  • etc...
GLSL passing data around

- **Application**
  - Triangles
  - Attributes
- **Vertex Program**
  - Varying parameters
- **Rasterizer**
  - Varying parameters
- **Fragment Program**
  - Depth
  - Color
- **Framebuffer**
- **Uniform Variables**
Uniform Variable

- A GLSL variable the user can specify value from the C/Java side.
- Its value is constant while drawing each vertex and pixel.
- Suitable for specifying
  - Material properties
  - Transformation matrices
  - Light sources
  - Textures
Declaring a Uniform Variable in GLSL

• Declare as a global variable (outside functions).
• Prefix the variable type with keyword “uniform.”
• Examples:

```glsl
// The values for these are initialized in the Java code!
uniform float shininess;
uniform vec3 color;
uniform mat4 model_transform;

void main()
{
    // Code here...
}
```
Caveats

• Uniform variables are shared between vertex and fragment shaders
  - Declare once in vertex shader and once more in fragment shader.

• As a result, types of uniform variables in vertex and fragment shaders must be consistent.

• Cannot have `uniform int x;` in vertex shader, but `uniform float x;` in fragment shader.

• Uniforms that are declared but not used are “optimized” out
  - OpenGL throws an error if you try to set a nonexistent uniform
Using Uniform Variables in the CS4620/4621 Framework

- Uniform variables are encapsulated by `GLUniform` class.
- Use `program.getUniform(<name>)` to get the instance (an integer) corresponding to the name.
- Set values by `GLUniform.set**(...)` methods.

```cpp
// In GLSL: uniform vec3 color;
program.use();
Vector3 c = new Vector3(1.0f, 0.5f, 1.0f);
GLUniform.set(program.getUniform("uGridColor"), c);

// In GLSL: uniform mat4 MVP;
// For matrices, use setST, not set! A boolean is provided
// for transposing.
GLUniform.setST(program.getUniform("VP"),
               camera.mViewProjection, false);
```
Attribute Variables

• A variable containing an attribute for a single vertex.
• Position, normal, texture coordinate, etc.
• Each time the shader is run, the attribute variables receive the values for the current vertex.
• These only appear in vertex shaders (Why?)
Attribute Mapping

• Attribute variables map to OpenGL buffers.
• OpenGL buffers have an index, GLSL attribute variables have a name.
• Must ensure the mapping from buffer indices to variable names is correct.
• In the provided framework:

```java
// Create a data buffer to fill in the attribute data
GLBuffer vertexPositions = new GLBuffer(BufferTarget.ARRAY_BUFFER, BufferUsageHint.STATIC_DRAW, true);

vertexPositions.setAsVertexVec3();

// Set vertexPositions, e.g. by reading in a Mesh
vertexPositions.useAsAttrib(program.getAttribute("vPos"));
```
Demo: Twisting
2D Twisting

- We transform vertices according to the following equation:

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} = \begin{bmatrix}
  \cos \left( t \sqrt{x^2 + y^2} \right) & -\sin \left( t \sqrt{x^2 + y^2} \right) \\
  \sin \left( t \sqrt{x^2 + y^2} \right) & \cos \left( t \sqrt{x^2 + y^2} \right)
\end{bmatrix} \begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]

where

- \((x,y)\) is the vertex position in object space.
- \((x',y')\) is the vertex position in clip space.
- \(t\) is the twisting factor, which is stored in the uniform variable “twisting”
TwistScreen (legacy commands)

```java
public void draw(GameTime gameTime) {
    ...
    program.use();

    int count = 100; //200;
    float size = 1.0f / count;
    GL11.glBegin(GL11.GL_QUADS); // GL11.GL_LINES // GL11.glBegin(GL11.GL_POINTS);

    for (int i = 0; i < count; i++)
        for (int j = 0; j < count; j++)
            {
                float x = -0.5f + i * size;
                float y = -0.5f + j * size;

                GL11.glVertex2f(x, y);
                GL11.glVertex2f(x + size, y);
                GL11.glVertex2f(x + size, y + size);
                GL11.glVertex2f(x, y + size);
            }
    GL11.glEnd();
    GLUniform.set(program.getUniform("color"), new Vector3(1, 0, 1));
    GL20.glUniform1f(program.getUniform("twisting"), 0); // (float)gameTime.total;

    program.unuse();
}
```
public void draw(GameTime gameTime) {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL2.GL_COLOR_BUFFER_BIT);

    program.use();

    // Set the uniforms
    GLUniform.set(program.getUniform("color"), color);
    GLUniform.set(program.getUniform("twisting"), 0.5f);

    // Set the attribute
    vertexPositions.useAsAttrib(program.getAttribute("in_Vertex"));

    glDrawElements(...); // Draw the mesh

    GLProgram.unuse();
}
#version 120

uniform float twisting;

void main()
{
    float angle = twisting * length(gl_Vertex.xy);
    float s = sin(angle);
    float c = cos(angle);
    gl_Position.x = c * gl_Vertex.x - s * gl_Vertex.y;
    gl_Position.y = s * gl_Vertex.x + c * gl_Vertex.y;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;
}
#version 120

uniform vec3 color;

void main()
{
    gl_FragColor = vec4(color, 1);
}
Varying Variables

• Interface between vertex and fragment shaders.
• Vertex shader outputs a value at each vertex, writing it to this variable.
• Fragment shader reads a value from the same variable, automatically interpolated to that fragment.
• No need to declare these in the Java program (Why?)
Declaring Varying Variables

• Declare as a global variable (outside functions).

• Syntax: varying <<type>> <<name>>;

• Example:

  ```
  varying vec3 color;
  
  void main()
  {
      // Some code here...
  }
  ```
Demo: Position as Color
Position as Color

• Compute the color of each fragment from its position in object space.

\[
\text{color} = \frac{\text{position} + (1,1,1)}{2}
\]
Vertex Shader Code

```glsl
#version 120

varying vec3 color;
attribute vec3 vPos;
uniform mat4 VP;

void main()
{
    gl_Position = VP * vec4(vPos,1);
    //color = (vec3(vPos) + vec3(1,1,1)) * 0.5;
    color = (vec3(gl_Position.xyz) + vec3(1,1,1)) * 0.5;
}
```
#version 120

varying vec3 color;

void main()
{
    gl_FragColor = vec4(color, 1);
}
Vertex Shader Code

#version 330

in vec3 vPos;

uniform mat4 VP;

out vec3 color;
out vec4 vPosition;

void main()
{
    vPosition = VP * vec4(vPos,1);
    color = (vec3(gl_Position.xyz) + vec3(1,1,1)) * 0.5;
}
Fragment Shader Code

#version 330

in vec3 color;
out vec3 vFragColor;

void main()
{
    vFragColor = vec4(color, 1);
}