Programming Graphics
Hardware

Computer Graphics, VT 2013
Lecture 10

Johan Nysjö
Centre for Image analysis
Swedish University of Agricultural Sciences
Uppsala University
Recap: The fixed-function OpenGL graphics pipeline

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels
Recap: The fixed-function OpenGL graphics pipeline

```c
#include <GL/glut.h>

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 1.0, 1.0); glVertex3f(-0.5, -0.5, 0.0);
    glColor3f(1.0, 1.0, 1.0); glVertex3f(0.5, -0.5, 0.0);
    glColor3f(1.0, 1.0, 1.0); glVertex3f(0.0, 0.5, 0.0);
    glEnd();
    glFlush();
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutCreateWindow("Triangle");
    glutDisplayFunc(display);
    glutMainLoop();
}
```
Modern shader-based OpenGL programming

- Since OpenGL 3.1, the fixed function pipeline has been eliminated and replaced with a **programmable graphics pipeline**

- This pipeline uses so-called **shader programs** to perform most of the vertex and fragment processing.

- Enables vertices and fragments to be processed in parallel on the GPU
Legacy OpenGL vs. modern shader-based OpenGL

The programmable OpenGL graphics pipeline
The OpenGL Shading Language (GLSL)

- A high-level, cross-platform shading language for real-time rendering
- Based on the C programming language. Uses similar naming conventions, data types, and control structures.
- Also provides vector and matrix types (vec2, vec3, mat4, etc), along with functions that operates on them
- Supports swizzling of vectors/matrices (e.g., v.xy = v.yx)
- See [http://www.opengl.org/documentation/glsl/](http://www.opengl.org/documentation/glsl/) for more information
Other real-time shading languages

• The High Level Shading Language (HLSL)
  - Developed by Microsoft for use with their Direct3D API
  - Similar to GLSL, but Windows-only
  - Frequently used in the games industry

• C for Graphics (Cg)
  - Developed by NVIDIA (predates GLSL and HLSL)
  - Direct3D/OpenGL cross-compability
  - More low-level than GLSL and HLSL. Awkward syntax.

• ARB assembly language (really low-level)
Shading languages for offline rendering

- Pixar RenderMan
- Houdini VEX Shading Language
- Maya Mental Ray Shading Language
- Gelato Shading Language
Vertex shaders

- Runs on the vertex processor and implements a
general-purpose programmable method for operating
on vertices. Used for, e.g.,
  - Vertex transformation
  - Normal transformation and normalization
  - Generating or transforming texture coordinates
  - Per-vertex lighting
  - Color material application
Vertex shader inputs

- Attributes
- Uniforms
- Samplers
- Shader program
Vertex shaders
Fragment shaders

- Runs on the fragment processor and implements a general-purpose programmable method for operating on fragments. Used for, e.g.,
  - Operating on interpolated varying data
  - Computing fragment color
  - Texture access
  - Texture application
  - Per-fragment lighting
  - Bump mapping
  - Image-based postprocessing effects
Fragment shader inputs

- Varying variables
- Uniforms
- Samplers
- Shader programs
Fragment shaders

Varying data is interpolated linearly across the primitive
Interpolation of varying data

- By default, all varying data that you pass from the vertex shader to the fragment shader will be linearly interpolated over the primitive (i.e., line or triangle)
- To prevent interpolation of an attribute, you can use the `flat` qualifier (only available in GLSL 1.30 and above)
The general process of creating and using shaders in OpenGL

1. Create a vertex shader object and a fragment shader object
2. Attach source code to the shader objects
3. Compile them
4. Create a program object
5. Attach the compiled shaders to the program object
6. Link the program (and check for errors)
7. Activate the program at drawing
Transforms in modern OpenGL

- Fixed-function pipeline functions such as glRotate, glFrustum, and gluLookAt are deprecated/removed since OpenGL 3.0
- The standard way of applying transforms in modern OpenGL is to construct them on the host CPU and pass them to the vertex shader as uniform variables
- You can either implement your own mathematics library (not recommended)...
- or (better) use one of the many existing third-party mathematics libraries for graphics programming
OpenGL Mathematics (GLM) library

- A header-only C++ mathematics library for graphics programming
- Provides vector and matrix datatypes, transforms, quaternions, noise functions, and much more
- Similar syntax as GLSL
- Link: http://glm.g-truc.net/
- Alternative mathematics libraries:
  - Eigen (a more heavy-weight linear algebra library)
  - CML
Example 1 – Simple triangle
Vertex shader

#version 130

attribute vec4 a_position;

void main() {
  gl_Position = a_position;
}
Fragment shader

#version 130

void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

Example 2 – Spinning RGB cube
Vertex shader

#version 130

uniform mat4 u_mvp;
attribute vec4 a_position;
varying vec4 v_color;

void main() {
    v_color.rgb = 0.5 * (a_position.xyz + 1.0);
    v_color.a = 1.0;
    gl_Position = u_mvp * a_position;
}
Fragment shader

#version 130

varying vec4 v_color;

void main() {
    gl_FragColor = v_color;
}
Example 3 – Per-fragment Blinn-Phong shading
Vertex shader

#version 130

uniform mat4 u_mv;
uniform mat4 u_mvp;
uniform vec3 u_light_position;
uniform mat3 u_normal_matrix;

attribute vec4 a_vertex;
attribute vec3 a_normal;

varying vec3 v_normal;
varying vec3 v_light_direction;
varying vec3 v_viewer;

void main()
{
    vec3 ec_pos = vec3(u_mv * a_vertex);
    v_normal = normalize(u_normal_matrix * a_normal);
    v_viewer = -ec_pos;
    v_light_direction = u_light_position - ec_pos;
    gl_Position = u_mvp * a_vertex;
}
# Fragment shader

```glsl
#version 130

varying vec3 v_normal;
varying vec3 v_light_direction;
varying vec3 v_viewer; // direction to the viewer across a triangle

void main() {
  vec3 N = normalize(v_normal);
  vec3 L = normalize(v_light_direction);
  vec3 V = normalize(v_viewer);
  vec3 H = normalize(L + V); // halfway vector

  // Compute diffuse, ambient, and specular terms

  gl_FragColor = vec4(diffuse_intensity * diffuse * diffuse_color +
                      ambient_intensity * ambient_color +
                      specular_intensity * specular * specular_color, 1.0);
}
```
Example 4 – Vertex displacement and analytic normals
#version 130

uniform mat4 u_mvp;
uniform mat4 u_mv;
uniform mat3 u_normal_matrix;
uniform float u_time;
uniform vec3 u_light_position;
in vec4 a_vertex;
out vec3 v_light_direction;
out vec3 v_viewer;
out vec2 v_xy;
const float PI = 3.141592;

void main() {
    // Compute new vertex position
    vec4 position = a_vertex;
    float amplitude = 0.1;
    float wavelength = 0.3;
    float wavenumber = 2.0 * PI * dot(position.xy, position.xy) / wavelength;
    float period = 1.0;
    float angular_frequency = 2.0 * PI * u_time / period;
    float phase = wavenumber - angular_frequency;
    position.z = amplitude * sin(phase);
    v_xy = a_vertex.xy;

    // Compute view and light vectors
    ...
    gl_Position = u_mvp * position;
}
# Fragment shader

```glsl
#version 130

uniform float u_time;
uniform mat3 u_normal_matrix;
in vec3 v_light_direction;
in vec3 v_viewer;
in vec2 v_xy;
const float PI = 3.141592;

void main() {
  // Compute analytic normal
  float amplitude = 0.05;
  float wavelength = 0.3;
  float wavenumber = 2.0 * PI * dot(v_xy, v_xy) / wavelength;
  float period = 1.0;
  float angular_frequency = 2.0 * PI * u_time / period;
  float phase = wavenumber - angular_frequency;
  float dfdx = 2.0 * PI * 2.0 * amplitude * v_xy.x * cos(phase) / wavelength;
  float dfdy = 2.0 * PI * 2.0 * amplitude * v_xy.y * cos(phase) / wavelength;
  vec3 xtangent = vec3(1.0, 0.0, dfdx);
  vec3 ytangent = vec3(0.0, 1.0, dfdy);
  vec3 normal = normalize(cross(xtangent, ytangent));

  // Compute diffuse, ambient, and specular terms with ...
  ... gl_FragColor = vec4(diffuse_intensity * diffuse * diffuse_color +
                          ambient_intensity * ambient_color +
                          specular_intensity * specular * specular_color, 1.0);
}
```
Example 5 – Texture mapping
Vertex shader

#version 130

uniform mat4 u_mvp;
uniform mat4 u_mv;
uniform vec3 u_light_position;
uniform mat3 u_normal_matrix;

in vec4 a_vertex;
in vec3 a_normal;
in vec2 a_texcoord;

out vec3 v_normal;
out vec3 v_viewer;
out vec3 v_light_direction;
out vec2 v_texcoord;

void main() {
    // Compute v_normal, v_viewer, and v_light_direction
    ...
    v_texcoord = a_texcoord;
    gl_Position = u_mvp * a_vertex;
}
Fragment shader

#version 130

uniform sampler2D u_texture0;
in vec3 v_normal;
in vec3 v_viewer; // direction to the viewer across a triangle
in vec3 v_light_direction;
in vec2 v_texcoord;
out vec4 color;

void main() {
    vec3 N = normalize(v_normal);
    vec3 L = normalize(v_light_direction);
    vec3 V = normalize(v_viewer);
    vec3 H = normalize(L + V); // halfway vector

    vec4 texture_color = texture(u_texture0, v_texcoord.ts);
    vec3 diffuse_color = texture_color.rgb;
    // Compute diffuse, ambient, and specular terms

    color = vec4(diffuse_intensity * diffuse * diffuse_color +
                  ambient_intensity * ambient_color +
                  specular_intensity * specular * specular_color, 1.0);
}
In conclusion

- Compared with classical fixed-function OpenGL, modern shader-based OpenGL programming is more low-level and has a steeper learning curve...
- but is also more powerful and flexible!
- Using programmable shaders, you can create effects that are difficult or impossible to create with fixed-function OpenGL
Web resources

• A few tutorials that does a good job of teaching modern OpenGL programming:
  - http://open.gl/
  - http://www.arcsynthesis.org/gltut/
  - http://tomdalling.com/blog/category/modern-opengl/
Books

- Real-Time Rendering (good second graphics book)
- OpenGL ES 2.0 Programming Guide (teaches modern shader-based OpenGL)
- OpenGL Shading Language (3rd Edition)
More GPU programming resources

• The GPU Gems series (a bit old, but free!)
  - GPU Gems 1
  - GPU Gems 2
  - GPU Gems 3

• The webpage for the Real-Time Rendering book contains links and references to lots of resources:
Lecture 12 (reserved session)

- I will go through a few more advanced shader programming concepts, like
  - Image-based postprocessing effects (e.g., edge-detection)
  - How to create and use procedural noise shaders
  - How to implement cube environment mapping in GLSL
Shader programming in lab 4

- In lab 4, you will be given code to
  - Load and create shader programs
  - Read volumetric CT data (foot.raw) from file into a 3D texture
  - Set up framebuffer objects (FBOs) for rendering to texture

- Your main task will be to implement the vertex and fragment shaders that are used to perform GPU-accelerated ray-casting

- The lab code uses GLSL 1.10 and OpenGL 2.1