Basic Graphics Programming

Graphics Pipeline
OpenGL API
Core and Compatibility Profiles
Primitives: Lines, Polygons
Attributes: Color

[Angel Ch. 2]
What is OpenGL

• A low-level graphics library (API) for 2D and 3D interactive graphics.
• Descendent of GL (from SGI)
• First version in 1992; now: 4.5 (August 2014)
• Managed by Khronos Group (non-profit consortium)
• API is governed by Architecture Review Board (part of Khronos)
Where is OpenGL used

- CAD
- Virtual reality
- Scientific visualization
- Flight simulation
- Video games
Graphics library (API)

- Intermediary between applications and graphics hardware

- Other popular APIs:
  - Direct3D (Microsoft)
  - OpenGL ES (embedded devices)
  - X3D (successor of VRML)
OpenGL is cross-platform

- Same code works with little/no modifications
- Windows: default implementation ships with OS
  Improved OpenGL: Nvidia or AMD drivers
- Linux: Mesa, a freeware implementation
  Improved OpenGL: Nvidia or AMD drivers
- Mac: ships with the OS
OpenGL is cross-platform

Include file (OpenGL Compatibility Profile):

```c
#if defined(WIN32) || defined(linux)
    #include <GL/gl.h>
    #include <GL/glu.h>
    #include <GL/glut.h>
#elif defined(__APPLE__)
    #include <OpenGL/gl.h>
    #include <OpenGL/glu.h>
    #include <GLUT/glut.h>
#endif
```
OpenGL is cross-platform

Include file (OpenGL Core Profile):

```c
#if defined(WIN32) || defined(linux)
    #include <GL/glew.h>
    #include <GL/glut.h>
#elif defined(__APPLE__)
    #include <OpenGL/gl3.h>
    #include <OpenGL/gl3ext.h>
    #include <GLUT/glut.h>
#endif
```
How does OpenGL work

From the programmer’s point of view:

1. Specify geometric objects

2. Describe object properties
   • Color
   • How objects reflect light
How does OpenGL work (continued)

3. Define how objects should be viewed
   • where is the camera
   • what type of camera

4. Specify light sources
   • where, what kind

5. Move camera or objects around for animation
The result
OpenGL is a state machine

State variables: vertex buffers, camera settings, textures, background color, hidden surface removal settings, the current shader program...

These variables (the *state*) then apply to every subsequent drawing command.

They persist until set to new values by the programmer.
OpenGL Library Organization

• **GL** (Graphics Library): core graphics capabilities
• **GLUT** (OpenGL Utility Toolkit): input and windowing
• **GLEW** (Extension Wrangler): removes OS dependencies
• **GLU** (OpenGL Utility Library; compatibility profile only): utilities on top of GL

![Diagram showing the relationship between OpenGL application program, GLEW, GL, GLUT, Graphics driver, Graphics card, and Computer monitor.]
Core vs Compatibility Profile

- **Core Profile:**
  - “Modern” OpenGL
  - Introduced in OpenGL 3.2 (August 2009)
  - Optimized in modern graphics drivers
  - Shader-based
  - Used in our homeworks

- **Compatibility Profile:**
  - “Classic” OpenGL
  - Supports the “old” (pre-3.2) OpenGL API
  - Fixed-function (non-shader) pipeline
  - Not as optimized as Core Profile
Graphics Pipeline

- Vertices
- Transformer
- Clipper
- Projector
- Rasterizer
- Pixels

Primitives + material properties → Translate → Rotate → Scale → Is it visible on screen? → 3D to 2D → Convert to pixels → Shown on the screen (framebuffer)
The Framebuffer

• Special memory on the graphics card

• Stores the current pixels to be displayed on the monitor

• Monitor has no storage capabilities

• The framebuffer is copied to the monitor at each refresh cycle
Rendering with OpenGL

- Application generates the geometric primitives (polygons, lines)
- System draws each one into the framebuffer
- Entire scene redrawn anew every frame
- Compare to: off-line rendering (e.g., Pixar Renderman, ray tracers)
The pipeline is implemented by OpenGL, graphics driver and the graphics hardware.

OpenGL programmer does not need to implement the pipeline.

However, pipeline is reconfigurable ➔ “shaders”
• Efficiently implementable in hardware (but not in software)

• Each stage can employ multiple specialized processors, working in parallel, busses between stages

• #processors per stage, bus bandwidths are fully tuned for typical graphics use

• Latency vs throughput
• Vertices in **world coordinates**

```c
void glVertex3f(GLfloat x, GLfloat y, GLfloat z)
```

– Vertex \((x, y, z)\) is sent down the pipeline.
– Function call then returns.

• Use **GLtype** for portability and consistency

• `glVertex{234}{sfid}[v](TYPE coords)`
Vertices (core profile)

- Vertices in **world coordinates**
- Store vertices into a Vertex Buffer Object (VBO)
- Upload the VBO to the GPU during program during program initialization (before rendering)
- OpenGL renders directly from the VBO
Transformer (compatibility profile)

- Transformer in **world coordinates**
- Must be set **before** object is drawn!
  
  \[
  \text{glRotatef}(45.0, 0.0, 0.0, -1.0);
  \]
  \[
  \text{glVertex2f}(1.0, 0.0);
  \]

- Complex [Angel Ch. 3]
Transformer (core profile)

- Transformer in world coordinates
- 4x4 matrix
- Created manually by the user
- Transmitted to the shader program before rendering
Clipper

- Mostly automatic (must set viewport)
Projector

- Complex transformation [Angel Ch. 4]

Orthographic

Perspective
Rasterizer

- Interesting algorithms [Angel Ch. 6]
- To window coordinates
- Antialiasing
Primitives (compatibility profile)

- Specified via vertices
- General schema

```c
glBegin(type);
    glVertex3f(x1, y1, z1);
    ...
    glVertex3f(xN, yN, zN);
glEnd();
```

- `type` determines interpretation of vertices
- Can use `glVertex2f(x, y)` in 2D
Example: Draw Square Outline (compatibility profile)

- **Type** = GL_LINE_LOOP

  ```
  glBegin(GL_LINE_LOOP);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(1.0, 1.0, 0.0);
  glVertex3f(0.0, 1.0, 0.0);
  glEnd();
  ```

- Calls to other functions are allowed between `glBegin(type)` and `glEnd();`
Primitives (core profile)

- Specified via vertices
- Stored in a Vertex Buffer Object (VBO)

```c
int numVertices = 300;
float vertices[3 * numVertices];
// (... fill the “vertices” array ...)
// create the VBO:
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```
Render Points and Line Segments (compatibility profile)

glBegin (GL_POINTS);
glVertex3f(…);

…
glVertex3f(…);
glEnd();
Render Points and Line Segments (core profile)

```c
glDrawArrays(GL_POINTS, 0, numVertices);
```
Main difference between the two profiles

Compatibility:

Rendering:
`glBegin(type);` 
`glVertex3f(x1, y1, z1);` 
`...` 
`glVertex3f(xN, yN, zN);` 
`glEnd();`

Core:

Initialization:
`int numVertices = 300;` 
`float vertices[3 * numVertices];` 
`// (… fill the “vertices” array …)`
`// create the VBO:`
`GLuint buffer;`
`glGenBuffers(1, &buffer);`
`glBindBuffer(GL_ARRAY_BUFFER, buffer);`
`glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);`

Rendering:
`glDrawArrays(GL_POINTS, 0, numVertices);`
Mixing core and compatibility profiles

- Windows, Linux:
  Can mix core and compatibility profile OpenGL commands
  ➔ can lead to confusion
    (is the specific OpenGL command optimized?)
  ➔ advantage: more flexible (can re-use old code)

- Mac:
  Can only choose one profile (in each application)
Polygons

• Polygons enclose an area

• Rendering of area (fill) depends on attributes
• All vertices must be in one plane in 3D
• GL_POLYGON and GL_QUADS are only available in the compatibility profile (removed in core profile since OpenGL 3.1)
Polygon Restrictions (relevant for compatibility profile only)

- OpenGL Polygons must be **simple**
- OpenGL Polygons must be **convex**

(a) simple, but not convex

(b) non-simple

(c) convex
Why Polygon Restrictions?

- Non-convex and non-simple polygons are expensive to process and render.
- Convexity and simplicity is expensive to test.
- Behavior of OpenGL implementation on disallowed polygons is “undefined.”
- Some tools in GLU for decomposing complex polygons (tessellation).
- Triangles are most efficient.
- Polygons removed since OpenGL 3.1.
Triangle Strips

- Efficiency in space and time
- Reduces visual artefacts
Attributes:
color, shading and reflection properties

• Set before primitives are drawn
• Remain in effect until changed!
Physics of Color

- Electromagnetic radiation
- Can see only a tiny piece of the spectrum
Color Filters

- Eye can perceive only 3 basic colors
- Computer screens designed accordingly

Source: Vos & Walraven
Color Spaces

• RGB (Red, Green, Blue)
  – Convenient for display
  – Can be unintuitive (3 floats in OpenGL)

• HSV (Hue, Saturation, Value)
  – Hue: what color
  – Saturation: how far away from gray
  – Value: how bright

• Other formats for movies and printing
RGB vs HSV

Gimp Color Picker

RGB vs HSV

Gimp Color Picker
Flat vs Smooth Shading

Flat Shading

Smooth Shading
Flat vs Smooth Shading

color of last vertex

each vertex separate color smoothly interpolated

Compatibility profile:
glShadeModel(GL_FLAT)

Compatibility profile:
glShadeModel(GL_SMOOTH)

Core profile: use interpolation qualifiers in the fragment shader
Viewport

- Determines clipping in window coordinates
- `glViewport(x, y, w, h)` (usually in reshape function)
Summary

1. Graphics pipeline
2. OpenGL API
3. Core and compatibility profiles
4. Primitives: vertices, lines, polygons
5. Attributes: color