Graphics Pipeline

Primitives: Points, Lines, Triangles [Angel Ch. 2]

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”

Graphics Pipeline

- Efficiently implementable in hardware (but not in software)
- Each stage can employ multiple specialized processors, working in parallel, buses between stages
- #processors per stage, bus bandwidths are fully tuned for typical graphics use
- Latency vs throughput
Vertices (compatibility profile)

- 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

```c
glVertex2f(1.0, 0.0);
``` 

- Complex [Angel Ch. 3]

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!

```c
glRotatef(45.0, 0.0, 0.0, -1.0);
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 viewing volume)

Projector

- Complex transformation [Angel Ch. 4]

  Orthographic
  
  Perspective
Rasterizer

- Interesting algorithms [Angel Ch. 6]
- To window coordinates
- Antialiasing

Geometric Primitives

- Suppose we have 8 vertices: \( p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7 \)
- Then, one can interpret them as:

![Diagram of 8 vertices]

- \( GL\_POINTS, GL\_LINES, GL\_TRIANGLES \) are examples of primitive type

Triangles

- Can be any shape or size
- Well-shaped triangles have advantages for numerical simulation
- Shape quality makes little difference for basic OpenGL rendering

Example: Draw Two Square Edges

- Type = GL\_LINES
  ```markdown
  glBegin(GL\_LINES);
  glVertex3f(0.0, 0.0, -1.0);
  glVertex3f(1.0, 0.0, -1.0);
  glVertex3f(1.0, 1.0, -1.0);
  glVertex3f(0.0, 1.0, -1.0);
  glEnd();
  ```
- Calls to other functions are allowed between `glBegin(type)` and `glEnd();`

Geometric Primitives (compatibility profile)

- Specified via vertices
- General schema
  ```markdown
  glBegin(type);
  glVertex3f(x1, y1, z1);
  ...
  glVertex3f(xN, yN, zN);
  glEnd();
  ```
- `type` determines interpretation of vertices
- Can use `glVertex2f(x,y)` in 2D

Geometric Primitives (core profile)

- Specified via vertices
- Stored in a Vertex Buffer Object (VBO)
  ```markdown
  int numVertices = 300;
  float vertices[3 * numVertices];
  // (… fill the “vertices” array …)
  // create the VBO:
  GLuint vbo;
  glGenBuffers(1, &vbo);
  glBindBuffer(GL\_ARRAY\_BUFFER, vbo);
  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);
  ```
Render Points and Line Segments (compatibility profile)

```
gBegin (GL_POINTS); // or GL_LINES to render lines
gVertex3f(…);
...
gVertex3f(…);
gEnd();
```

Main difference between the two profiles

**Compatibility:**
```
Initialization:
int numVertices = 300;
float vertices[3 * numVertices];
// (...fill the "vertices" array ...)
// create the VBO:
finish(vbo);  // GL_ARRAY_BUFFER

Rendering:
gBegin(type);  // GL_POINTS or GL_LINES
  glVertex3f(x1, y1, z1);
  ...  // add more vertices
  glVertex3f(xN, yN, zN);
gEnd();
```

**Core:**
```
int numVertices = 50000;
float * vertices = (float*) malloc(sizeof(float) * 3 * numVertices);
...
finish(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Common Bug
```
int numVertices = 50000;
float * vertices = (float*) malloc(sizeof(float) * 3 * numVertices);
finish(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

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

Summary

1. Graphics pipeline
2. Primitives: vertices, lines, triangles