CSCI 420 Computer Graphics
Lecture 3

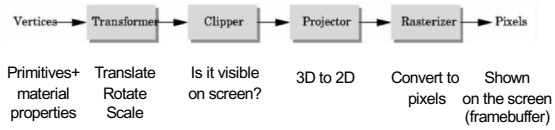
Graphics Pipeline

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

Jernej Barbic
University of Southern California

1

Graphics Pipeline



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

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

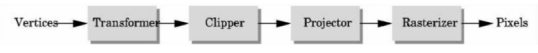
3

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)

4

The pipeline is implemented by OpenGL, graphics driver and the graphics hardware

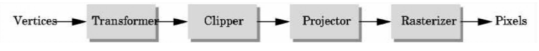


OpenGL programmer does not need to implement the pipeline.

However, pipeline is reconfigurable
➔ "shaders"

5

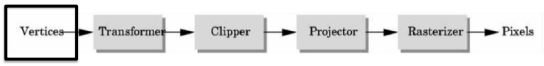
Graphics Pipeline



- 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

6

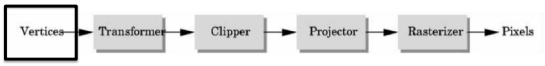
Vertices (compatibility profile)



- Vertices in world coordinates
  void glVertex3f(GLfloat x, GLfloat y, GLfloat z)
  – Vertex (x, y, z) is sent down the pipeline.
  – Function call then returns.
- Use GL*type* for portability and consistency
- glVertex{234}{sfid}[v](*TYPE coords*)

7

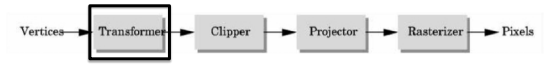
Vertices (core profile)



- 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

8

Transformer (compatibility profile)



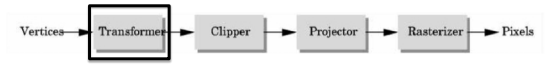
- Transformer in world coordinates
- Must be set before object is drawn!

```
glRotatef(45.0, 0.0, 0.0, -1.0);
glVertex2f(1.0, 0.0);
```

- Complex [Angel Ch. 3]

9

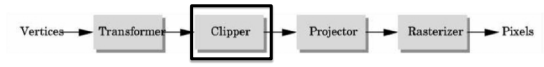
Transformer (core profile)



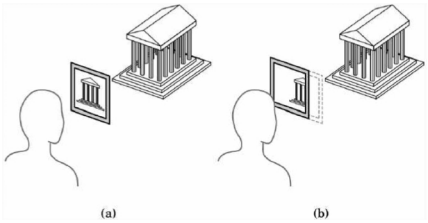
- Transformer in world coordinates
- 4x4 matrix
- Created manually by the user
- Transmitted to the shader program before rendering

10

Clipper

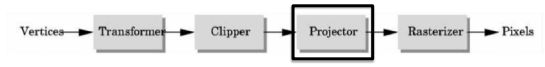


- Mostly automatic (must set viewing volume)



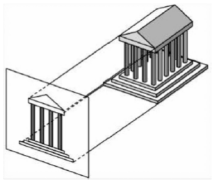
11

Projector

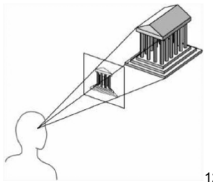


- Complex transformation [Angel Ch. 4]

Orthographic

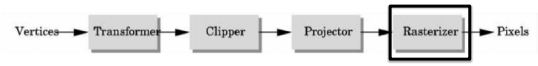


Perspective

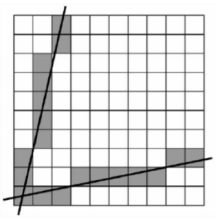


12

Rasterizer



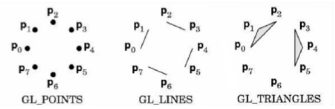
- Interesting algorithms [Angel Ch. 6]
- To window coordinates
- Antialiasing



13

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:

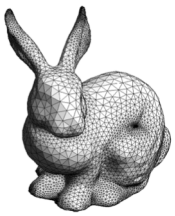


- GL_POINTS, GL_LINES, GL_TRIANGLES are examples of primitive *type*

14

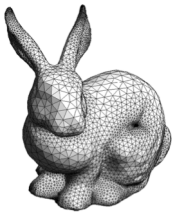
Triangles

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



15

Geometric Primitives (compatibility profile)



- Specified via vertices
- General schema

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

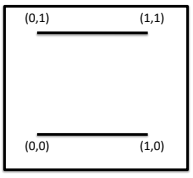
- *type* determines interpretation of vertices
- Can use glVertex2f(x,y) in 2D

16

Example: Draw Two Square Edges (compatibility profile)

- *Type* = GL_LINES

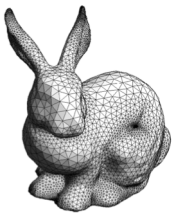
```
glBegin(GL_LINES);
   glVertex3f(0.0, 0.0, -1.0);
   glVertex3f(1.0, 0.0, -1.0);
   glVertex3f(1.0, 1.0, -1.0);
   glVertex3f(0.0, 1.0, -1.0);
glEnd();
```



- Calls to other functions are allowed between glBegin(*type*) and glEnd();

17

Geometric Primitives (core profile)

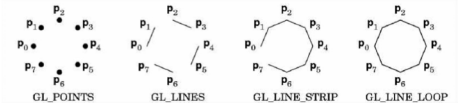


- Specified via vertices
- Stored in a Vertex Buffer Object (VBO)

```
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);
```

18

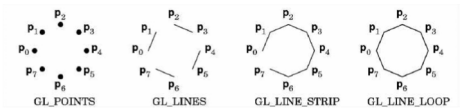
Render Points and Line Segments (compatibility profile)




```
glBegin (GL_POINTS); // or GL_LINES to render lines
  glVertex3f(…);
  …
  glVertex3f(…);
glEnd();
```

19

Render Points and Line Segments (core profile)




```
glDrawArrays(GL_POINTS, 0, numVertices); // render points
glDrawArrays(GL_LINES, 0, numVertices); // render lines
```

20

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 vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
  sizeof(vertices), vertices, GL_STATIC_DRAW);

Rendering:
glDrawArrays(type, 0, numVertices);
```

21

Common Bug

```
int numVertices = 50000;
float * vertices = (float*) malloc (sizeof(float) * 3 * numVertices);
…
glBufferData(GL_ARRAY_BUFFER,
  sizeof(vertices), vertices, GL_STATIC_DRAW);
```

What is wrong?

22

Common Bug

```
int numVertices = 50000;
float * vertices = (float*) malloc (sizeof(float) * 3 * numVertices);
…
glBufferData(GL_ARRAY_BUFFER,
  sizeof(vertices), vertices, GL_STATIC_DRAW);
```

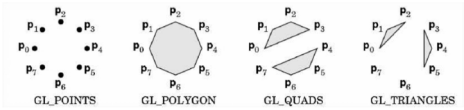
```
glBufferData(GL_ARRAY_BUFFER,
  sizeof(float) * 3 * numVertices, vertices, GL_STATIC_DRAW);
```

✓

23

Polygons

- Polygons enclose an area

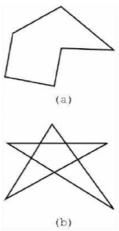


- Rendering of area (fill) depends on attributes
- All vertices must be in one plane in 3D
- GL_POLYGON and GL_QUADS are only available in the compatibility profile (removed in core profile since OpenGL 3.1)

24

Polygon Restrictions (relevant for compatibility profile only)

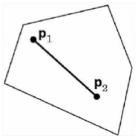
- OpenGL Polygons must be simple
- OpenGL Polygons must be convex



(a) simple, but not convex

(b) non-simple

(c) convex



25

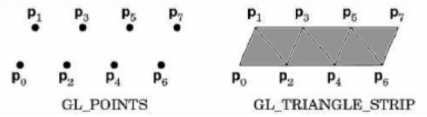
Why Polygon Restrictions?

- 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

26

Triangle Strips

- Efficiency in space and time
- Reduces visual artefacts



27

Summary

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

28