Graphics Programming

- We will focus on OpenGL
- We will discuss "new" OpenGL (Version 4.5 in Red Book)
- C++ knowledge assumed

OpenGL History

- OpenGL 1.0 (1992)
- Immediate-mode graphics
- Graphic primitives (e.g. draw a line) were specified in applications
- Immediately passed to hardware for display
- Redisplay requires all primitives to be resent (and redraw)

OpenGL History

- OpenGL 2.0 (2004)
- Introduced OpenGL Shading Language (GLSL)
- Can write own shaders and use GPUs
- Retained-mode graphics: geometry information can be stored or retained in GPU memory

OpenGL History

- OpenGL 3.0 (2008)
- “Old” style OpenGL deprecated
- OpenGL 4.0 (2010)

Pipeline Architecture

- Tasks are split up into multiple stages in the pipeline
- Different stages can operate at the same time on different data (increasing throughput)
- Main steps:
  - Vertex processing
  - Clipping and primitive assembly
  - Rasterization
  - Fragment processing

Vertex Processing

- A vertex is a location in space (e.g. a point)
- Geometric objects (primitives) are specified by vertices
- Coordinate transformations are done here:
  - translation, rotation, scaling
  - world coordinates to camera coordinates
  - projection onto display plane
  - done by matrix multiplication
- Assignment of colour to a vertex is also done here
- Output is the location, colour and other attributes of each vertex

Clipping and Primitive Assembly

- The output device (e.g. screen) cannot display the entire world
- Clipping is done to remove objects that are not in the clipping volume (some can be partially visible)
- Clipping cannot be done individually by vertices.
- Sets of vertices are assembled into primitives (e.g. lines, triangles) before clipping can be done
- Output is a set of primitives that will appear in the display

Rasterization

- Primitives are converted to pixels
- The rasterizer determines which pixels in the frame buffer are affected by the primitive (e.g. lines, triangles)
- Output is a set of fragments: each fragment is a pixel together with information (e.g. colour, location, depth)
- Fragments are used to update the corresponding pixels in the framebuffer

Fragment Processing

- In simple scenes, each pixel has a corresponding fragment which is used to display that pixel
- However some pixels may have multiple fragments (e.g. 3D scenes with many objects and different depth)
- Some fragments may be blocked and not visible
- Fragments can also be blended (e.g. transparent objects)

Programmable Pipelines

- In “old” OpenGL (1.0), some of the stages are fixed and cannot be changed.
- Modern OpenGL allows stages to be customized.
- Both vertex and fragment processing can be customized

OpenGL Shading Language (GLSL)

- A C-like language for shaders
- Designed to execute directly on GPU (instead of CPU)
- Loaded at run time, not compile time
- Need to set up input/output to communicate with main program

Interface

- There are a number of different GUI libraries for OpenGL applications.
- We will use GLUT in this course
- See example code from previous editions of textbook
- The Red Book uses GLFW
- Set up window, display function, callback for input, etc.
- Run display loop
- Event-based programming: display, reshape, keyboard, mouse, idle, etc.
- Multiple viewports

Workflow Step 1: Vertex Array Objects

- The first step is to set up geometric primitives (e.g. objects) to display.
- Geometric primitives are specified by vertices.
- Many GPUs can only render (quickly) points, lines, and triangles—these are the only primitives supported.
- The vertices (positions and other attributes) are sent to the GPU using a vertex array object.
- The function `glDrawArrays` is used to draw the primitives specified by the vertices (a mode parameter is used to interpret the primitive).
- `GL_POINTS` just draws individual points

Vertex Array Objects

- Identified by an non-negative integer ID (for communicating with OpenGL).

```
enum VAO_IDS { Triangles, Lines, NumVAOs };
```

- These VAO needs to be created with `glGenVertexArrays` or `glCreateVertexArrays`.
- An array of `GLuint` should be used to store the IDs returned.
- Then a VAO has to be bound as the current object `glBindVertexArray`.
- Bind with 0 and use `glDeleteVertexArrays` when done.

Buffer Objects

- VAOs only give IDs. Buffer objects are needed to pass data to GPU.
- `glCreateBuffers` or `glGenBuffers`: returns IDs of buffer objects. Delete with `glDeleteBuffers`.
- Need to bind it with `GLBindBuffer`: for now use `GL_ARRAY_BUFFER` as target. There are other types of buffers.
- Loading data: `gl(Named)BufferStorage`, `gl(Named)BufferData`

Coordinate System

Vertices have to be specified in some coordinate system.

- 3D coordinates: x, y, and z.
- For 2D coordinates, keep z constant.
- All coordinates a floating-point number between -1 and 1 to be visible
- There can be many coordinate systems in an application:
  - Model coordinates
  - Object/World coordinates
  - Eye/Camera coordinates
  - Clip coordinates
  - Window coordinates

Line Primitives

Given an array of vertices:

- `GL_LINES`: many line segments specified by points 0 and 1, 2 and 3, 4 and 5, etc.
- `GL_LINE_STRIP`: adjacent vertices specify lines
- `GL_LINE_LOOP`: similar to line strip but last vertex connected to first one.

Triangle Primitives

- `GL_TRIANGLES`: each group of 3 vertices specify a triangle
- `GL_TRIANGLE_STRIP`: each group of 3 adjacent vertices specify a triangle
- `GL_TRIANGLE_FAN`: each triangle is specified by the first vertex and 2 adjacent vertices
- Orientation are used to determine “front” or “back”
- More complex shapes (e.g. polygons) are specified by triangles
- e.g. a solid circle (disc) can be approximated with a fan

Shaders

- Shaders have to be loaded using custom code
- See example code
- Need to set up input and output to communicate with main program

Shaders

- Shader variables: input/output to main program
- Can use `layout` to specify a position to communicate with main program
- Special output variables (e.g. `gl_Position`)
- Connect data to shaders with `glVertexAttribPointer` and `glEnableVertexAttribArray`.

Colours

- Specified by RGB values (each between 0 to 1)
- RGBA: a 4th channel called alpha is used to allow for transparency or opacity: $0 =$ transparent, $1 =$ opaque.

Viewing

By default:

- Only coordinates within -1 to 1 are visible
- Orthographic projection: what you would see if you place the camera infinitely far from objects
- Camera is placed at origin, looking in the negative z direction
- Takes a point (x, y, z) and projects it into $(x, y, 0)$
- Objects "behind" the camera can also be seen
- The clippping rectangle is what can be seen (between -1 and 1)
- Aspect ratio of clipping rectangle vs. aspect ratio of viewport can lead to distortion

Double Buffering

Useful especially for animiation:

- If the drawing occurs on the visible screen, there may be flicker
- Use 2 buffers: one for current display (front) and one for drawing (back)
- When drawing is complete, swap the buffers (`glutSwapBuffers`)
- `glutPostRedisplay` can be used to force redraw
- Make use of Idle event callback to recalculate and update, uninstall callback if animation should stop.