05 - The OpenGL Graphics Pipeline
Previously…
Viewing Transformation

- object space
- camera space
- screen space

- model
- camera
- projection
- viewport

- world space
- canonical view volume
How to do it?

- Specialized hardware — Graphics Processing Unit (GPU)
- APIs to interact with the hardware
  - OpenGL
  - DirectX
  - Metal/Mantle/Vulkan
https://open.gl

• This slide set is based on the excellent tutorial available at https://open.gl

• Many thanks to:
  • Toby Rufinus
  • Eric Engeström
  • Elliott Sales de Andrade
  • Aaron Hamilton
Context Creation

• Before you can draw anything you need to:

  • Open a window (i.e. ask the OS to give you access to a portion of the screen)

  • Initialize the OpenGL API and assign the available screen space to it

  • This is a technical step and it is heavily dependent on the operating system and on the hardware
Window Manager

• There are many libraries that take care of this for you, hiding all the complexity and providing a cross-platform and cross-hardware interface

• We are going to use GLFW (http://www.glfw.org)

• A window manager usually provides an event management system
```c
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);

    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);

        updateScene();

        drawGraphics();
        presentGraphics();
    }

    return 0;
}
```
Event Processing

• Callback mechanisms

• For every event (keypressed, mouse motion) you need to write a function that handles it

• The functions are registered in glfw, that will call them whenever the corresponding event happens

• We will see an example later on
Calling the API — glew

// Specify prototype of function
typedef void (*GENBUFFERS) (GLsizei, GLuint*);

// Load address of function and assign it to a function pointer
GENBUFFERS glGenBuffers = (GENBUFFERS)wglGetProcAddress("glGenBuffers");
// or Linux:
GENBUFFERS glGenBuffers = (GENBUFFERS)glXGetProcAddress((const GLubyte *)"glGenBuffers");
// or OSX:
GENBUFFERS glGenBuffers = (GENBUFFERS)NSGLGetProcAddress("glGenBuffers");

// Call function as normal
GLuint buffer;
glGenBuffers(1, &buffer);
CPU and GPU memory

You write code here …

CPU

- 2-12 cores
- Cache (1-12Mb)
- 4-64 Gb RAM
- Memory Buffer with a Mesh

PCI express

GPU

- 1k+ cores
- Cache (<1kb/core)
- 4-16 Gb STRUCTURED RAM

… which acts here.
“Modern” OpenGL

• OpenGL 1.0 was released in 1992 - historically very rigid and with a high level API that was dealing with all the pipeline steps

• “Modern” OpenGL usually refers to the lightweight OpenGL 3 (2008)

• Barebone, but adaptable: since each API call has a fixed and high overhead, the library was design to minimize them

• Much easier to use, but you need to understand how it works completely!
OpenGL pipeline

{ vertices }
and attributes
(colors, normals)

optional!
CPU and GPU memory

You write code here …

CPU

2-12 cores

Cache (1-12Mb)

4-64 Gb RAM

… which acts here.

GPU

1k+ cores

Cache (~1kb/core)

4-16 Gb STRUCTURED RAM

PCI express
Double/Triple Buffering

1. Single-buffering
   - Video memory: clear draw A clear draw B clear draw C clear draw D clear
echo draw E

2. Double-buffering
   - Buffer: clear draw A clear draw B clear draw C clear draw D clear
echo draw E
   - Video memory: copy A copy B copy C copy D copy E

3. Triple-buffering
   - Buffer 1: clear draw A clear draw C clear draw D clear draw E
echo draw F
   - Buffer 2: clear draw B clear draw D clear draw E clear draw F
echo draw G
   - Video memory: copy A copy B copy C copy D copy E copy F copy G

By Cmglee - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=20161108
OpenGL pipeline

{ vertices }
and attributes
(colors, normals)

optional!
Vertex Input

• You have to send to the GPU a set of vertex attributes:
  • World Coordinates
  • Color
  • Normal

• You can pass as many as you want, but remember that bandwidth is precious, you want to send only what is needed
Device Coordinates

• Only the vertices in the canonical cube will be shown. The cube will be stretched to fill all available screen space.

• All vertices must be passed in one instruction, in a single contiguous block of memory

• This matrix resides in CPU memory!

```c
float vertices[] = {
    0.0f,  0.5f, // Vertex 1 (X, Y)
    0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f // Vertex 3 (X, Y)
};
```
GPU Memory

• It is faster and closer to the GPU cores

• You want to minimize the number of transfers, the more you can reuse between two frames the better

• The throughput is very high but unfortunately also the latency. Try to send as much data as possible with one single instruction

• You need to manually manage the GPU memory and you can have memory leaks on the GPU too, be careful
Vertex Buffer Object

• Strange name for something simple, a chunk of memory in the GPU space

GLuint vbo;
glGenBuffers(1, &vbo); // Generate 1 buffer

• vbo contains a "opengl reference" to the buffer, which is simply an integer

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

• Now the content of vertices is uploaded to the GPU memory
Buffer Types

• You can provide hints to the API depending on the type of data that you are uploading:

```c
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

• For small applications this does not have a significant effect, but it is important to get it right, since the performance difference is massive

• **GL_STATIC_DRAW**: The vertex data will be uploaded once and drawn many times (e.g. the world).
• **GL_DYNAMIC_DRAW**: The vertex data will be changed from time to time, but drawn many times more than that.
• **GL_STREAM_DRAW**: The vertex data will change almost every time it’s drawn (e.g. mouse cursor).
Shaders

• The name is historical — they were introduced to allow customization of the shading step of the traditional graphics pipeline

• In modern OpenGL, shaders are general purpose functions that will be executed in parallel on the GPU

• They are written in a special C-like language called GLSL. You send a string containing your program to the OpenGL API: the OpenGL driver compiles it on-the-fly and uploads it to the GPU

• Extremely powerful and simple to write!

• The only “annoyance” is that the graphic card vendors do not fully respect the standard, leading to possible compatibility problems (it is getting better, but problems still happen)
Shaders

{ vertices }
and attributes
(colors, normals)

optional!

Geometric Transformations

Depth Test

Tests and Blending

Fragment shader

Lighting

Barycentric Interpolation

Rasterization

Geometry shader

Shape assembly

Vertex shader
Vertex Shader

- The vertex shader is a program on the graphic card that processes each vertex and its attributes as they appear in the vertex array.
- Its duty is to output the final vertex position in device coordinates and to output any data the fragment shader requires.
- All transformations from world to device coordinates happen here.
Vertex Shader

- **object space**
- **camera space**
- **screen space**

- **model**
- **camera**
- **projection**

- **world space**
- **canonical view volume**
- **viewport**
A simple vertex shader

```cpp
float vertices[] = {
    0.0f,  0.5f, // Vertex 1 (X, Y)
    0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};

#version 150

in vec2 position;

void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
}
```

`gl_Position` is a special keyword
Shaders

{ vertices } and attributes (colors, normals)

optional!

Vertex shader → Shape assembly → Geometry shader

Geometric Transformations

Tests and Blending → Fragment shader → Barycentric Interpolation

Depth Test → Lighting → Rasterization

CSCI-GA.2270-001 - Computer Graphics - Daniele Panozzo
Geometry Shader?

Each block only needs a position and a type, the actual geometry is implicit and can be created on the fly!
Shaders

{ vertices } and attributes (colors, normals)

optional!
Fragment Shader

• The output from the vertex shader is interpolated over all the pixels on the screen covered by a primitive.

• These pixels are called fragments and this is what the fragment shader operates on.

• Just like the vertex shader it has one mandatory output, the final color of a fragment. It’s up to you to write the code for computing this color from all the attributes that you attached to the vertices.
CSCI-GA.2270-001 - Computer Graphics - Daniele Panozzo

Vertex Shader  Fragment Shader + Tests

object space  camera space  screen space

model  camera  projection  viewport

world space  canonical view volume
A simple fragment shader

```cpp
float vertices[] = {
    0.0f, 0.5f, // Vertex 1 (X, Y)
    0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};

#version 150

out vec4 outColor;

void main()
{
    outColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

The colors inside a shader are between 0.0 and 1.0
Compiling Shaders

const GLchar* vertex_shader =
    "#version 150 core
    "in vec2 position;
    "void main()"
    "{gl_Position = vec4(position, 0.0, 1.0);}"

const GLchar* fragment_shader =
    "#version 150 core
    "out vec4 outColor;
    "uniform vec3 triangleColor;"
    "void main()"
    "{ outColor = vec4(triangleColor, 1.0); }"

GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertex_shader, NULL);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragment_shader, NULL);
glCompileShader(fragmentShader);
Almost there, OpenGL program

• The two shaders needs to be combined into a program

```c
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
```

We need to connect the program with our input data and map the output to a memory buffer or to the screen
Almost there, OpenGL program

• The two shaders needs to be combined into a program

```c
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
```

• The output of the fragment shader connected with the default frame buffer ("0")

```c
glBindFragDataLocation(shaderProgram, 0, "outColor");
```

• The program is now ready to be linked and activated

```c
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
```
Connecting VBOs to the program

// This is our set of vertices
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// This is our vertex shader
#version 150
in vec2 position;
void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
}

• We want to connect vbo to the “position” slot, so that the program
  will know where to find the vertex positions
Connecting VBOs to the program

- Bind the program
  
  ```
  glUseProgram(shaderProgram);
  ```
- Query the program to find the address of “position”
  
  ```
  GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
  ```
- Bind the vbo we want to connect
  
  ```
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  ```
- Connect the VBO to the “position” slot
  
  ```
  glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
  ```
- Activate the attribute
  
  ```
  glEnableVertexAttribArray(posAttrib);
  ```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);

• posAttrib is the index of the attribute

• 2 indicates that the VBOs has two attributes per vertex (x and y)

• GL_FLOAT indicates that the VBO stores single precision floating point numbers

• 0 is the stride, or how many bytes are between each position attribute in the array

• the last 0 is the offset, or how many bytes from the start of the array the attribute occurs

• There is no type checking, if you do a mistake here you will render whatever is in the memory address that you specified and possibly crash the graphic driver
Vertex Array Objects - VAO

• Changing OpenGL program is easy, but it would be annoying to have to redo all the connections between VBOs and input every time

• A VAO stores these connections and allow to easily switch between different configurations

```c
GLuint vao;
glGenVertexArrays(1, &vao);

glBindVertexArray(vao);
```

• After the bind, all the subsequent calls to glVertexAttribPointer will be stored in the VAO

• Tip: You must have a binded VAO at all times. If you do not have it, glVertexAttribPointer will ignore the bind without giving you an error :-)

We are all set!

• Now that we prepared the program, linked the input array and set the output to the frame buffer we are ready to draw our triangle

```c
glDrawArrays(GL_TRIANGLES, 0, 3);
```

• The first parameter is the type of primitive (in this case, it draws a triangle after each set of three vertices is processed)

• 0 is the **offset**, i.e. how many vertices it has to skip from the beginning of the VBOs

• 3 is the number of **VERTICES** (not primitives) that it should process
If everything was done correctly…
If everything was done correctly…

Our current pipeline does not consider view transformations!

(Recommended Exercise: how would you change the current shaders to take the size of the window into account and always draw a equilateral triangle?)
Uniforms

• Uniform are values that are constant for the entire scene, i.e. they are not attached to vertices

• They correspond to global variables in C/C++

• All vertices and all fragments will see the same value for the same uniform

• For example, let’s use a uniform to store the triangle color (which is now hardcoded in our shader)
Uniforms

Old Fragment Shader

```glsl
#version 150

out vec4 outColor;

void main()
{
    outColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

New Fragment Shader

```glsl
#version 150

uniform vec3 triangleColor;

out vec4 outColor;

void main()
{
    outColor = vec4(triangleColor, 1.0);
}
```

GLint uniColor = glGetUniformLocation(shaderProgram, "triangleColor");

`glUniform3f(uniColor, 1.0f, 0.0f, 0.0f);`
Recap

- Compile, link and enable an OpenGL program (vertex + fragment shader)
- Connects the output of the fragment shader to the frame buffer
- Connect the VBOs to the input slots of the vertex shader using a VAO
- Assign the uniform parameters (if you use them in the shaders)
- Clear the framebuffer
- Draw the primitives
- Swap the framebuffer to make the newly rendered frame visible
Some tips

• Start from a **WORKING** OpenGL application and do your changes incrementally. It is really easy to do mistakes and debugging OpenGL is frustrating.

• When an OpenGL error happens, you have to “probe” the API to see what went wrong (see function check_gl_error() in Helpers.h).

• It is possible that your application will produce slightly different results on different machines. I suggest to do all the development on one machine (or at least on one single GPU brand). If you have two GPUs, be careful to not let the OS switch between them.
Assignment 2

• Let’s take a look at Assignment 2 and at the provided code

• We did not cover everything that you need for Assignment 2 yet, but you can already start with Task 1.1, 1.2, and 1.3

• Start early, the assignment looks easy, but it is not
References

https://open.gl — Main reference

4th Edition by Steve Marschner, Peter Shirley

Chapter 17