Drawing

Motivation

- We want to render something to the screen
- This requires us to understand how the GPU handles rendering

Basic Idea

- We simulate motion by drawing scene many times a second
- Each time, scene may be moved a bit
- Gives illusion of smooth continuous motion

Rendering

- Our first render: We'll just draw a single point

Buffers

- Concept: Buffer
  - GPU is physically separate from CPU
  - Some GPU's have separate RAM from main memory
  - So we need to push data from CPU memory to GPU memory
  - Buffer = chunk of memory accessible to GPU

Buffers

- Basic pattern for creating buffer:
  - glGenBuffer
  - glBindBuffer
  - glBufferData \
- This gen/bind/set pattern is used repeatedly in GL

glGenBuffer

- GL uses integers to refer to *resources*
  - Resource = Buffer, texture, ...
- We need to "generate" a unique buffer ID when we want to create a new buffer
- Complication: GL was designed as a C-based API
  - So its use looks a little un-Pythonic

glGenBuffer

- We first need to use a special Python interface: The *array*
  - import array
- `tmp = array.array(“I”, [0] )`
  - Create an array that holds integers
  - Initialize it with a single value equal to zero

glGenBuffer

- Now we can call the GL function:

  ```
  glGenBuffers(1,tmp)
  buffID = tmp[0]
  ```

- The “1” tells how many buffers we want to create

glBindBuffer

- Our program will have *many* buffers (eventually)
- How does GL know which one we want to work with?
- Concept: Binding point
  - A binding point can be associated with one (and only one) buffer
  - We *bind* a buffer to a binding point when we want to work with it
  - We *unbind* the buffer when we're done with it

glBindBuffer

- Code: glBindBuffer(GL_ARRAY_BUFFER, buffID)
  - GL_ARRAY_BUFFER is the binding point we’re using
  - This binding point is for vertex data
  - Second parameter is the buffer we want to work with

glBufferData

- Now we can ship data from CPU to GPU
- We'll start off with a simple 2D point
- Again, we must use the array type since we need to push a blob of raw memory around

```
tmp = array.array("f", [0,0] ).tobytes()
glBufferData( GL_ARRAY_BUFFER, len(tmp), tmp, GL_STATIC_DRAW)
```

glBufferData

- First parameter = which binding point we're using
- Second = size of data, in bytes
- Third = the data itself
- Fourth = usage hint
  - GL_STATIC_DRAW = We specify the data once, but we'll draw with it many times

Done!

- We're done with this buffer, so we should unbind it: glBindBuffer(GL_ARRAY_BUFFER, 0)
  - Zero = special value
  - Tells GL we want nothing attached to the binding point

Note

- Buffers are so commonly used that we should factor it all out into a separate, reusable class
- (Do in class)

```
#Buffer.py
class Buffer:
    def __init__(self, dataAsFloatArray ):
        ...
    def bind(self, bindingPoint):
        ...
```

Using It

- Suppose we create the buffer like so:
  `myBuffer = Buffer( array.array("f",[0,0]) )`
- Now we're ready to use it... Right?
  - Nope.

Using It

- A buffer doesn't provide enough information for the GPU to use it
- Remember, it's just a blob of bytes
- How does the GPU know how to interpret this blob?
  - It doesn't!

Using It

- *We* know it's a list of 2D float values
- But the GPU does not
- So we must tell the GPU how information is organized in the buffer

VAO

- Concept: Vertex Array Object (VAO)
- Allows us to tell GPU how one or more buffers are to be used
- Follows same basic pattern as buffers:
  - glGenVertexArrays
  - glBindVertexArray
  - Set information
  - Unbind

glGenVertexArrays

- Same idea as with buffers: Generate a VAO

```
tmp = array.array("I",[0])
glGenVertexArrays(1,tmp)
vao = tmp[0]
```

glBindVertexArray

- Same idea as with buffers: Bind the VAO so we can work with it: glBindVertexArray(vao)
  - Tells GL we want to work with this particular vao
  - There's no binding point specified here – just the vao we want to work with

Data

- Now we will specify data
- This is where we tell GL about the buffer's layout and that we want to use the buffer

```
myBuffer.bind(GL_ARRAY_BUFFER)
glEnableVertexAttribArray(0)
glVertexAttribPointer( 0, 2, GL_FLOAT, False, 2*4, 0 )
```

Explanation

- myBuffer.bind(GL_ARRAY_BUFFER)
- We bind the buffer we want to use
- Easy!

Explanation

- glEnableVertexAttribArray(0)
- Then we enable the attribute array
- *Vertex puller* can grab data from up to ≈16 buffers at a time
  - Think of puller as having 16 “pipes”
  - Data comes down a pipe and gets used for rendering
- This function call tells GL pipe zero should be enabled
  - GL records this fact in the vao

Explanation

- glVertexAttribPointer(0, 2, GL_FLOAT, False, 2*4, 0)
- Finally, we tell GL where to find the data
  - First parameter = Which pipe we're talking about
  - Second = how many items per vertex
  - Third = what type each item is
  - Fourth = auto-normalize (this is nearly always false)
  - Fifth = size of each data item, in bytes
  - Sixth = where item starts in buffer (byte offset)
- All this information is recorded in the vao
- In the background, this also takes currently bound buffer and notes that in vao too

Unbind

- Finally, we unbind: glBindVertexArray(0)
  - Since we're done with this buffer

Draw

- OK, *now* we can draw something, right?

Draw

- OK, *now* we can draw something, right?
- Nope.

Shader

- We need to tell GPU what to do with the data
- So far, we've just told the GPU
  - What our data is (the raw bytes)
  - How it's organized

GPU

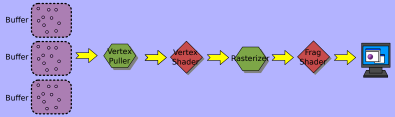
- Structure of GPU: Highly parallel computer
  - Designed to process lots of data at the same time
  - Modern GPU’s: Easily in the *teraflop* range
- We must instruct GPU how to do its work
- This requires that we write a small program (“shader”) to direct all that parallel processing

GLSL

- Shader is written in a C-like language called GLSL
  - GL Shading Language

Pipeline

- There are actually two shaders to write
- GPU pipeline:



Explanation

- Fixed functionality
  - Hardwired into silicon
  - Fast
  - Limited configurability

Fixed Functionality

- Vertex puller: Grabs data from one or more buffers
  - Dispatches to vertex shader
- Rasterizer
  - Takes output of VS
  - Determines which pixels are covered

Shaders

- This is where most of our attention is focused
- Two shaders of interest: Vertex shader, Fragment shader

Vertex Shader

- Runs once per vertex
- Takes input from the buffers we specified in VAO
- GL expects it will output *screen space location* of that vertex
- If we're drawing points, GL also expects it to say how many pixels are covered by that point

Fragment Shader

- Runs once per pixel that is covered by primitive we’re drawing
- Takes input from the buffers we specified in VAO
- GL expects it will output *screen space location* of that vertex

Vertex Shader

```
#version 430
layout(location=0) in vec2 position;
void main(){
    gl_Position = vec4( position.xy, -1, 1 );
    gl_PointSize = 1;
}
```

- Explain this in-class...

Fragment Shader

```
#version 430
out vec4 color;
void main(){
    color = vec4(1,1,1,1);
}
```

- Explain this in class

GLSL

- Quick syntax overview:
  - Every statement must end with a semicolon
  - Indentation is not significant
    - Blocks are denoted with { }
  - All variables (except builtins) must be *declared* before use
    - Declaration: type, variable name, semicolon

Shaders

- Shaders are just ordinary text files
- We can write with any editor we'd like
  - But must save as plain text!
- Customary to save with .vs or .fs suffix, but you don't have to
- We must write code to load data, push to GPU

Pattern

- We create a *program object*
  - Acts like container for our shaders
- Then: For each shader we:
  - Create a shader object
  - Specify shader code
  - Compile shader code
  - Attach shader to program object
- Finally, we *link* the shaders together
- And now we can *use* the program

Code

- We encapsulate shader operations into their own class:Program.py

Draw

- Can we finally draw something?

Draw

- Can we finally draw something?
- Yes!

Drawing

- To draw:
  - Make sure our program is active
  - Clear the screen
  - Bind the VAO
  - Call glDrawArrays()

Draw

- glDrawArrays(GL_POINTS,0,1)
  - First = Type of primitive to draw
    - Points, lines, or triangles
  - Second = Where in buffer to start (0=at beginning)
  - Third = How many vertices to draw

Result

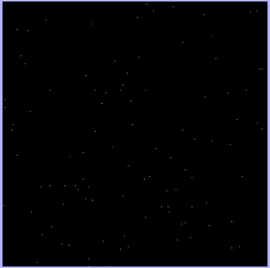
Wow!

Explanation

- GL screen space: -1...1 in x, y, and z
- z doesn't come into play (yet)
- We put our point at (0,0), so it's in the center of the screen.

Assignment

- Create and draw a *star field*
  - Bonus [+33%]: Make the stars different sizes
    - This will require a bit of independent research on your part
    - Hint: glEnable(GL_PROGRAM_POINT_SIZE)
  - Possibilities:
    - Create a second input for each vertex giving size.
    - Each VS invocation has access to global gl_VertexID (0,1,2,...). Use it.



Sources

- Khronos Group. OpenGL 4 quick reference card. http://www.khronos.org

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org