Uniforms

Motivation

- We often have data that applies to rendering of an entire object
- Need some way to send it to the GPU
- Solution: *Uniforms*

Name

- Called uniform because the value is the same (uniform) for all items of a single draw operation (glDrawArrays or glDrawElements)
- Can use from either VS or FS
- GL allows us to define a *uniform buffer*
- This is a block of RAM on GPU that holds uniform data

Buffer

- Since it's a buffer, it requires the same gen/bind/data operations that we've been doing
- We can just use our buffer class: ubo = Buffer(...initial data...)
- We'll discuss how we determine initial data in a moment

GLSL

- On the shader side, we declare and use uniforms like so:

```
#version 430
layout( std140, row_major ) uniform Uniforms {
    float foo;
    vec4 bar;
    ...more items follow...
};
...in or out variable declarations...
void main(){
    if( foo == 0.0 ){
        ...
    }
}
```

Question

- What’s the ‘std140’ in the layout()?
- Refers to *packing rules*
  - We can request GL use one of several formats
  - std140 = Rules defined as of GLSL 1.40

Question

- What’s the ‘row_major’ in the layout()?
- Matrices can be in row or column major
- Most programming languages use row_major

Preliminaries

- We need to alter the Buffer type a bit
- Add a function:

```
def bindBase(self, bindingPoint, index):
    glBindBufferBase(bindingPoint, index, self.buffID)
```

- In GL, some buffer binding points are *indexed*
- We can attach different buffers to different indices within that binding point
- If that sounds confusing – don’t worry about it
  - We can follow a “cookbook” approach for the few times when we need this

Constructor

- We will also change the constructor to give some flexibility

```
class Buffer:
    def  __init__(self, arrayOfData,  usage=GL_STATIC_DRAW, size=
       None ):
        tmp = array.array("I", [0] )
        glGenBuffers(1,tmp)
        self.buffID = tmp[0]
        glBindBuffer( GL_ARRAY_BUFFER, self.buffID )
        if arrayOfData == None:
            glBufferData( GL_ARRAY_BUFFER, size, None, usage )
        else:
            tmp = arrayOfData.tobytes()
            glBufferData( GL_ARRAY_BUFFER, len(tmp), tmp, usage)
        glBindBuffer( GL_ARRAY_BUFFER, 0 )
    ...rest of the code as before...
```

Program

- Next, the changes to Program
- These are a bit more extensive
- Concept: *static data*
  - Eventually, we'll have several Program's
  - Uniform data will be shared by all of them

Declarations

- Class-level (static) variables:

```
class Program:
    uniforms = {}
    ubo = None
    def __init__(...):
        ...
```

Constructor

- Add some code at end of constructor:

  ```
  if Program.ubo == None:
      Program.setupUniforms(self.prog)
  ```

- Notice how we call static function: Use class name before the dot

setupUniforms

- Define the setupUniforms method
- Notice: It does *not* take ‘self’ as an argument
- And it has the staticmethod *decorator:*

```
class Program:
    ...stuff....
    @staticmethod
    def setupUniforms(prog):
        ...
```

Code

- Get information about how many uniforms we have:

  ```
  tmp = array.array("I",[0])
  glGetProgramiv(prog,GL_ACTIVE_UNIFORMS, tmp);
  numuniforms = tmp[0]
  ```

Query

- For the next functions, we’re going to need to tell GL which uniforms we want information about
- We want to get info about all of them
- So we create an array with the requisite entries

```
tmp = []
for i in range(numuniforms):
    tmp.append(i)
uniformsToQuery = array.array“(”I,tmp)
```

Query

- Now to get the information

```
offsets = array.array("I",[0]*numuniforms)
sizes = array.array("I",[0]*numuniforms)
types = array.array("I",[0]*numuniforms)
glGetActiveUniformsiv(prog, numuniforms,
    uniformsToQuery, GL_UNIFORM_OFFSET, offsets);
glGetActiveUniformsiv(prog, numuniforms,
    uniformsToQuery, GL_UNIFORM_SIZE, sizes);
glGetActiveUniformsiv(prog, numuniforms,
    uniformsToQuery, GL_UNIFORM_TYPE, types);
```

Result

- Now, offsets[i] has the offset of uniform i
- And sizes[i] has the size of uniform i
  - For now, this will always be 1
  - Later, we might use arrays
    - In this case, size won't be 1
- And types[i] has the type of uniform i

Sizes

- We’ll define a dictionary so we can save ourselves some typing
- This holds the size of the types, in bytes:

```
sizeForType = {
    GL_FLOAT_VEC4: 4*4,
    GL_FLOAT_VEC3: 3*4,
    GL_FLOAT_VEC2: 2*4,
    GL_FLOAT: 1*4,
    GL_INT: 1*4
}
```

Uniforms

- Loop over the uniforms, get their names, and record some information:

```
nameBytes = bytearray(256)
Program.totalUniformBytes = 0
for i in range(numuniforms):
    glGetActiveUniformName(prog, i,
        len(nameBytes), tmp, nameBytes);
    nameLen = tmp[0]
    name = nameBytes[:nameLen].decode()
    if offsets[i] != 0xffffffff:
        assert sizes[i] == 1    #sanity check
        numBytes = sizeForType[types[i]]
        Program.uniforms[name] = ( offsets[i], numBytes, types[i]  )
        end = offsets[i] + numBytes
        if end > Program.totalUniformBytes:
            Program.totalUniformBytes = end
```

Finish Up

- Now we are ready to finish up
- We need to create a couple of things:
  - A block of CPU memory that will hold the uniforms
  - A variable to tell us where that block is
  - A block of GPU memory to hold the uniforms
    - This is a buffer

Finish Up

- Here’s what we need to do:

```
Program.uboBackingMemory = ctypes.create_string_buffer(Program.
    totalUniformBytes)
Program.uboBackingAddress = ctypes.addressof(Program.
    uboBackingMemory)
Program.ubo = Buffer(
    data=None,
    size=Program.totalUniformBytes,
    usage=GL_DYNAMIC_DRAW )
```

Finish Up

- One last thing: GL expects the uniform buffer to be bound to an *indexed binding point*
- So now we'll use the function we created previously: Program.ubo.bindBase(GL_UNIFORM_BUFFER,0)
- That's it for the setup work!

Setting Uniform

- How to set a uniform?
  - We’ll write a function for that
  - Uniform data is shared by all Program objects, so we use a static method
- So:

```
class Program:
    ...
    @staticmethod
    def setUniform(name,value):
        ...
```

Operations

- First, we query our uniform dictionary for the information about the uniform in question:
  offset,numBytes,typ = Program.uniforms[name]

Convenience

- We can’t pass raw Python numbers to GL
- So we’ll do a quick check and wrap scalars in single-element arrays:

```
if typ == GL_FLOAT:
    value = array.array("f",[value])
elif typ == GL_INT:
    value = array.array("I",[value])
```

Operations

- Now we convert the value we have to a blob of raw bytes:
  b = value.tobytes()
- And we verify it has the expected size

```
if len(b) != numBytes:
    raise RuntimeError("Type mismatch when setting uniform '"+
        name+"': Got "+str(type(value)))
```

Operations

- Finally, we copy the data to the CPU buffer:

```
dst =  ctypes.c_void_p(Program.uboBackingAddress+offset)
ctypes.memmove( dst, b, numBytes )
```

- That’s it!

Updating

- Note though, that we've just been setting data in CPU memory
- This doesn't affect GPU memory at all!
- For that, we need to write a function to push data from CPU memory → GPU memory

Updating

- Another method in Program:

```
@staticmethod
def updateUniforms():
    glBufferSubData(GL_UNIFORM_BUFFER, 0,
        Program.totalUniformBytes,
        Program.uboBackingMemory)
```

Pitfall

- Since VS and FS are compiled separately, we need to specify the uniform block in both of them
  - And it needs to be identical in both of them
- We will also need to use information from the uniform block in CPU code too
- Retyping all of this is bad
  - DRY: Don't Repeat Yourself
  - WET: Write Everything Twice
- It's better to be DRY than WET

Solution

- We’ll specify the uniforms in a text file
- Our Program object will read that data in and insert it into the code that’s fed to glShaderSource

Uniforms

- Create a file in the shaders folder called “uniforms.txt”
- For this example, we’ll have just one uniform:

```
vec2 translation;
```

- That one line is all that goes in the file
- Also remove the #version line from the VS and FS files

Code

- Now we want to change Program’s compile() function:

```
def compile(self,fname,shaderType):
    s = glCreateShader(shaderType)
    shaderdata = open(os.path.join("shaders",fname)).read()
    uniformdata = open(os.path.join("shaders","uniforms.txt")).
       read()
    src = [ "#version 430\n",
        "layout( std140, row_major ) uniform Uniforms {\n",
        "#line 1\n",
        uniformdata,
        "};\n",
        "#line 1\n",
        shaderdata ]
    glShaderSource(s, len(src), src, None )
    glCompileShader(s)
    ...rest as before...
```

Putting It All Together

- The whole thing:
  - Buffer.py
  - globalVars.py
  - main.py
  - Program.py
  - shaders/vs.txt
  - shaders/fs.txt
  - shaders/uniforms.txt

Finally

- We’re ready for a simple demo
- In main.py:

```
translation=vec2(0,0)
translationSign=vec2(1,1)
def update(elapsed):
    translation.x += translationSign.x * 0.5 * elapsed
    if translation.x > 1:
        translationSign.x = -abs(translationSign.x)
    if translation.x < -1:
        translationSign.x = abs(translationSign.x)
    Program.setUniform("translation",translation)
    Program.updateUniforms()
```

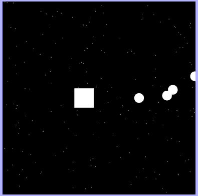
Vertex Shader

- Vertex shader:

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

Assignment

- Create a “Hero” object (which can just be a square).
  - A,D = move left, right; W = jump
    - Note to self: Describe state machines in class.
  - Space = Fire bullet (hexagon) from the player’s position. It should move in whatever direction (left or right) the hero last moved.
- For bonus [+10%], S should cause the hero to crouch. This means the square gets smaller vertically.
  - But: You can’t create a second vertex buffer...You’ll have to find some other way of making the hero crouch.
- You might want the math3d.py file from the class website



Sources

- Ehhh... None that I recall...

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