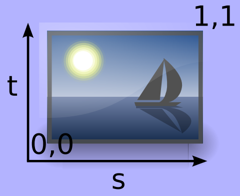
Textures

Motivation

- Monochromatic objects = Boring
- Texturing: Process of using image to affect object color
- What do we need?
  - Texture: 2D image
  - Object being textured
  - Mapping: Tells what parts of 2D texture go on object

Texture Mapping

- OpenGL defines texture axes s and t
- DirectX is similar, but uses names u and v
- Texture coordinate *always* from 0...1
  - Independent of texture size in pixels
- Vocabulary: *Texel* vs. *Pixel*
- Note: GL says (0,0) is at lower left corner; DX says it's at upper left



Requirements

- We need several things
- We'll tackle them one at a time...

Setup

- An *image* to use as a texture
- A *texture object* which holds the data itself
- A *sampler object* which says how to sample data from the image
- *Texture coordinates* to tell where to sample data from
- A *shader* which gives instructions to tie it all together

Image

- First, the image
- Images stored as sequences of bytes
  - Byte = 8 bits = 0...255
  - RGB vs. BGR order
  - Transparency: RGBA or BGRA

Image

- To load images, we have a Python class that interfaces with the C routines for image load/store
  - Get it from the class webpage
  - On Windows/Linux: Can load JPEG or PNG
  - On Mac: Can only load PNG

Setup

- An *image* to use as a texture: Done!
- A *texture object* which holds the data itself
- A *sampler object* which says how to sample data from the image
- *Texture coordinates* to tell where to sample data from
- A *shader* which gives instructions to tie it all together

Texture

- We'll create a class for working with texture data
  - We'll actually create several classes
  - There are many different types of textures in GL
  - So we'll need several different types to work with them

Base Class

- The granddaddy of them all...

```
class Texture:
    def __init__(self, typ):
        self.type = typ
        self.tex = None
    def bind(self,unit):
        glActiveTexture(GL_TEXTURE0 + unit)
        glBindTexture(self.type,self.tex)
    def unbind(self,unit):
        glActiveTexture(GL_TEXTURE0 + unit)
        glBindTexture(self.type,0)
```

Explanation

- Texture type (self.type): What kind of texture it is
  - For now, we only use GL_TEXTURE_2D_ARRAY

Explanation

- Texture name (self.tex): GL uses integers to "name" textures
- We intend to subclass Texture, so we rely on the subclass to produce the name

Explanation

- Bind: Texture units
  - GL has several texture units
  - Each one can have a texture associated with it
  - We select which unit we want with glActiveTexture
  - Then we can associate a texture object using glBindTexture
- Unbind: The “name” 0 is special
  - Means “no texture”

Next

- Next: We will make a more specialized class for 2D textures:

```
class Texture2DArray(Texture):
    def __init__(self,w,h,slices):
        Texture.__init__(self,GL_TEXTURE_2D_ARRAY)
        self.w=w
        self.h=h
        self.slices=slices
```

Texture2DArray

- Notice the superclass constructor call
- Concept: Width and height of texture: Pretty simple
- Concept: Slices

Finally

- Now we can make the class we'll use directly:

```
class ImageTexture2DArray(Texture2DArray):
    def __init__(self, *files):
        ...
```

  - Notice the *variadic* argument
- What now?

First

- First task: We will have one or more images to push over to GL
- We will push the data all at once
- So we need to collect them together in a big blob

```
membuf = io.BytesIO()
w=None
h=None
slices=0
```

- Don't forget to import the io module
- BytesIO works like a file, but it buffers its data in RAM

Second

- Look at each file and load it, accumulating all data to the buffer
- We might want to support zip files too:

```
for fname in files:
    if fname.endswith(".png") or fname.endswith(".jpg"):
        ...part 1 code here...
    elif fname.endswith(".ora") or fname.endswith(".zip"):
        ...part 2 code here...
    else:
        raise RuntimeError("Cannot read file "+fname)
```

- This uses Python's built-in zipfile module
  - Make sure to import it!

Part 1

- To read a PNG or JPEG file: This is the "part 1 code":

```
with open(fname,"rb") as fp:
    tmp = fp.read()
pw,ph,fmt,pix = image.decode(tmp)
pix = image.flipY(pw,ph,pix)
if w == None:
    w=pw
    h=ph
else:
    if w != pw or h != ph:
        raise RuntimeError("Size mismatch")
slices += 1
membuf.write(pix)
```

Part 2

- To read a zip file, we make use of Python's zipfile module. This is the part 2 code:

```
z = zipfile.ZipFile(fname)
for n in sorted(z.namelist()):
    if n.endswith(".png") or n.endswith(".jpg"):
        tmp = z.open(n).read()
        pw,ph,fmt,pix = image.decode(tmp)
        pix = image.flipY(pw,ph,pix)
        if w == None:
            w=pw; h=ph
        else:
            if w != pw or h != ph:
                raise RuntimeError("Size mismatch")
        slices+=1
        membuf.write(pix)
```

Texture

- Now we're ready to push the data to GL
- For that, we'll need a texture object

Operations

- Textures: General API is similar to how buffers work:
  - Generate texture
  - Bind it to tell GL we want to work with it
  - Push data from CPU to GPU

First

- Create the texture object:
```
tmp = array.array("I",[0])
glGenTextures(1,tmp)
self.tex = tmp[0]
```

This is where we set superclass's 'tex' attribute

Next

- Make this texture object the active one
- Use texture unit zero:
  self.bind(0)

Data

- Specify the data:
  glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA, w,h,slices, 0, GL_RGBA, GL_UNSIGNED_BYTE, membuf.getbuffer())
- Parameters:
  - Type of texture
  - Mip level (we'll explain this later)
  - Internal format
  - Size (width, height, slices)
  - Border
  - Incoming data format
  - Incoming data type
  - Incoming data

Done!

- We're done!
- We can clean up by ensuring our texture isn't active: self.unbind(0)

The Whole Thing

- Here's the whole thing: [ImageTexture2DArray.py]
- The Texture2D.py and Texture2DArray.py are simple enough that you can just copy/paste them from the previous slides

Setup

- An *image* to use as a texture: Done!
- A *texture object* which holds the data itself: Done!
- A *sampler object* which says how to sample data from the image
- *Texture coordinates* to tell where to sample data from
- A *shader* which gives instructions to tie it all together

Sampler

- We now need a sampler object: Tells GL how to pull data from the texture

```
class Sampler:
    def __init__(self):
        tmp = array.array("I",[0])
        glGenSamplers(1,tmp)
        self.samp = tmp[0]
        glSamplerParameteri( self.samp,
            GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE)
        glSamplerParameteri( self.samp,
            GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE)
        glSamplerParameteri( self.samp,
            GL_TEXTURE_MAG_FILTER, GL_NEAREST)
        glSamplerParameteri( self.samp,
            GL_TEXTURE_MIN_FILTER, GL_NEAREST)
    def bind(self,unit):
        glBindSampler(unit, self.samp )
```


- Discuss: Difference between GL_NEAREST and GL_LINEAR

Setup

- An *image* to use as a texture: Done!
- A *texture object* which holds the data itself: Done!
- A *sampler object* which says how to sample data from the image: Done
- *Texture coordinates* to tell where to sample data from
- A *shader* which gives instructions to tie it all together

Setup

- Recall how we set up our position buffer for an object (in this case, a square)...

First

- We define vertex coordinates and indices to make the square:
  vbuff = Buffer(array.array("f", [-1,-1, 1,-1, 1,1, -1,1])
  ibuff = Buffer(array.array("I", [0,1,2, 0,2,3]))
- Suppose we want to texture this square

Now

- We'll also define a buffer with texture coordinates:
  tbuff = Buffer(array.array("f", [0,0, 1,0, 1,1, 0,1]))

Setup

- Recall: A VAO allows us to tell GL "use this set of buffers for a draw operation"
- Previously, we did the following:
  - Generate a new VAO
  - Bind it to tell GL we want to work with it
  - Associate the vertex and (maybe) index buffer with it
  - Activate vertex puller for slot zero
  - Tell GL about the data format for slot zero

Next

- We need to tweak things a bit
- Here's the code for setting up the VAO for a textured square

```
vbuff = Buffer( array.array("f",[
    -1,-1,   1,-1,   1,1,  -1,1 ]))
tbuff = Buffer( array.array("f", [
    0,0,     1,0,    1,1,   0,1 ] ))
ibuff = Buffer(array.array("I",[ 0,1,2,   0,2,3 ]))
tmp = array.array("I",[0])
glGenVertexArrays(1,tmp)
vao = tmp[0]
glBindVertexArray(vao)
ibuff.bind(GL_ELEMENT_ARRAY_BUFFER)
vbuff.bind(GL_ARRAY_BUFFER)
glEnableVertexAttribArray(0)
glVertexAttribPointer( 0, 2, GL_FLOAT, False, 2*4, 0 )
tbuff.bind(GL_ARRAY_BUFFER)
glEnableVertexAttribArray(1)
glVertexAttribPointer( 1, 2, GL_FLOAT, False, 2*4, 0 )
glBindVertexArray(0)
```

Note

- When you do glVertexAttribPointer, whichever buffer is currently bound to the GL_ARRAY_BUFFER “hook” is associated with that particular vertex puller slot
- That’s why we first bound vbuff, did glVertexAttribPointer(0, ...) and then bound tbuff and did glVertexAttribPointer(1, ...)
  - This associated vbuff with puller slot 0 and tbuff with puller slot 1

Setup

- An *image* to use as a texture: Done!
- A *texture object* which holds the data itself : Done!
- A *sampler object* which says how to sample data from the image : Done!
- *Texture coordinates* to tell where to sample data from : Done!
- A *shader* which gives instructions to tie it all together

Shader

- Vertex shader previously looked like this:

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

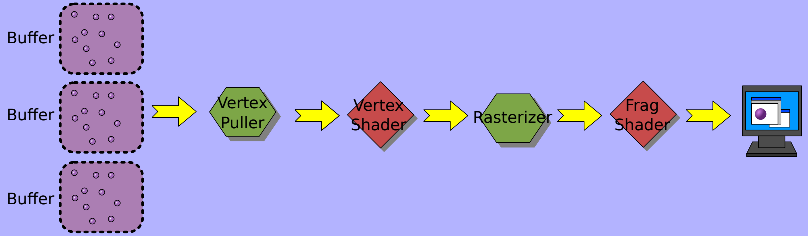
Shader

- We add a new input:

```
layout(location=0) in vec2 position;
layout(location=1) in vec2 texCoord;  //<---- new
void main(){
    gl_Position = vec4( position.xy, -1, 1 );
    gl_PointSize = 1;
}
```

Pipeline

- Recall how the GPU pipeline was structured:



Pipeline

- VS runs first, once per vertex of triangle
- Then rasterizer determines which pixels are covered
- And runs FS once for each pixel
- But: VS can pass data to FS

Passing Data

- Declare global variable in VS with tag “out”
- Declare global variable with same name in FS with tag “in”
- Whatever VS writes will be available to FS

VS

```
layout(location=0) in vec2 position;
layout(location=1) in vec2 texCoord;
out vec2 v_texCoord;
void main(){
    gl_Position = vec4( position.xy, -1, 1 );
    v_texCoord = texCoord;
}
```

FS

```
in vec2 v_texCoord;
out vec4 color;
void main(){
    ??? what goes here ???
}
```

Question

- Suppose system is going to draw a triangle
- VS runs three times: Once for each vertex
- Each VS execution produces a screen space coordinate (written to gl_Position) and a texture coordinate (written to v_texCoord)
- Rasterizer takes screen space coordinates and determines which pixels are covered
- FS runs once for each pixel to tell system what color goes at that pixel
- What value will FS get for v_texCoord?

Weighted Average

- GPU takes weighted average of VS out's for each pixel
- Ex: If we have vertices A, B, C: The closer we are to A, the more A has a "voice" in what v_texCoord should be

FS

- We can now finish the FS:

```
in vec2 v_texCoord;
layout(binding=0) uniform sampler2DArray tex;
out vec4 color;
void main(){
    color = texture(tex, vec3(v_texCoord,0) );
}
```

Explanation

- binding $\rightarrow$ Tells which texture unit we're using
- texture(): Reads from texture
  - First argument: The texture to read from
  - Second argument: A vec3
    - x = horizontal coordinate
    - y = vertical coordinate
    - z = slice number (0...n-1)

CPU Code

- Finally, we have our CPU side code
- In our main setup() function:

```
samp = Sampler()
samp.bind(0)
```

CPU Code

- In our various objects:

```
class Something:
    tex = None
    def __init__(self):
        if Something.tex == None:
            Something.tex = ImageTexture2DArray( "foo.png" )
        ...
```

CPU Code

- In our various objects' draw methods:

```
class Something:
    ...
    def draw(self):
        Something.tex.bind(0)    #must match binding=nnn in shader
        ...bind vao, set uniforms, and draw as before...
```

Pitfall

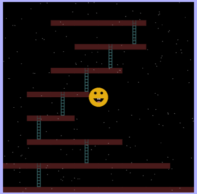
- GL assumes that each image row has a *stride* which is divisible by 4
  - If all our images are RGBA, no problem
  - If all our images have a width that is multiple of 4, no problem.
- Otherwise, we need to do this in main.py setup() (before you create any textures):

```
glPixelStorei(GL_PACK_ALIGNMENT,1)
glPixelStorei(GL_UNPACK_ALIGNMENT,1)
```

- You'll know if this is the case because your textures will look "sheared"

Assignment

- Read in a text file that contains a Tiled level description in XML+CSV format
- You might also want these four tile textures
- Retain the functionality from the previous lab (The hero doesn't need to respect the map position. Yet.)
- The stars should still be there and they should be white and stationary, as before
- You'll need to make a bullet texture



Sources

- Sampler Object. OpenGL Wiki.
  https://www.khronos.org/opengl/wiki/Sampler_Object

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