last time

- put back pipeline figure
- today will be very "codey"

OpenGL API

- library of routines to control graphics
- calls to compile and load shaders
- calls to load vertex data to vertex buffers
- calls to load textures
- draw calls
- calls to set various options in the pipeline

system issues

- cross platform
- the alternative would be DirectX graphics
  - dominant for PC games
  - not cross platform
- we will use 3.0
- we use the *glew* library to access the API
- needs include and library files, and installed drivers (see web page)

glut

- library of functions to talk with the windowing system
- open up windows
- glut can inform you when some "event" occurs
  - mousemove, buttonpress, windowresize, redraw needed
- you register callback functions with glut
  - the callback function is called when the event occurs
  - and passed relevant info (ex. the mouse location)
- cross platform
  - for real applications, you might use a platform dependent library.

glsl

- gl shading language
- you write small programs to be executed for each vertex.
- you write small programs to be executed for each fragment
- you tell openGL to compile/link/load these "shaders"
  - they operate in parallel on the vertices and fragments
- competitors
  - microsoft's hlsl:

    * dominant for pc games
    * only works with directX
  - nvidia's cg
    * almost identical to hlsl
    * also works with openGL
    * need cgGL library

code

- we use c++
- (double) d = 1./2;
  - see our primer.

main pattern for openGL resources

- note: this stuff is a bit annoying and confusing.
- openGL will provide us with storage for a few kinds of resources
  - shader programs, vertex buffers, textures
- we need to ask openGL for any such resource (AKA openGL object)
  - glCreateShader, glGenBuffer, glGenTexture,
  - openGL will return us with a "handle" (AKA object name), which allows us later to refer to this object with openGL using openGL calls.
    * the handle is of type GLuint
- when we are done, we need to tell openGL to free up the object with a glDelete* call

openGL resources in C++

- in C++ the clean way to do resource management is to always wrap each single resource request in its own class
  - the constructor calls the glGet*, or glCreate*
  - the destructor calls the glDelete*
- we store instances of these object in our own variables
- whenever one of our objects goes out of scope (no longer accessible by our program), the destructor is automatically called, which guarantees the resource release
- we do not allow our objects to get copied, so there is no issues of knowing when the resource can get deleted.
- lets look at glSupport.h
- note that we cannot make any GL calls until our program has called glutCreateWindow. so we cannot have any of these objects as global variables.
  - we can only have global pointers to such objects, and then construct new objects in the body of our code.

manipulating the openGL resources

- we need to put our data in these resources.
- we may need to change certain special settings for how the resource will be used.
- for glPrograms, this is done using special glCalls, and the appropriate handles.
- for buffers and textures we need to "bind" the particular resource to an openGL "target"
  - GL_ARRAY_BUFFER, and GL_TEXTURE_2D
- then we make openGL calls

- if needed, these calls also use the target name (but not the object name)

main

- initializes lots of stuff
- communicates with openGL API by loading glew.
- hands off all control to glut
  - glut will call back our own functions when needed to do updating and drawing

initsGlutState

- turns on glut
- requests a window with color, depth, and "double buffering"
- registers the names of our callback functions
  - we will look at them soon

initGLState

- sets some openGL control states
  - background color
  - plumbing for r/w of framebuffer
  - modern color space

initShaders

- note the use of some global pointers
  - we need globals since glut controls the computational flow
- dive down:

SquareShaderState

struct

- has GLProgram (construction gets openGL resources).
- has handles to the variables in our program
- dive down:

LoadSquareShader

- reads and complies the files (we will look at these shader files later)
- we have our own functions (defined in our own glSupport.h) to read the shader files and pass them to OpenGL
- gets "gl handles" to the input variables with the shown names in the shaders so we can pass info to them
  - the inputs are attribute and uniform shader variables
  - handles are really just integers identifiers

safe

calls (defined in our own glSupport.h)

- are our own wrappers around gl functions that won't cause an error if we try to set a variable that was optimized away for unuse
  - simplifies the code during shader development
- tells gl to use the variable named fragColor as the output of the fragment shader

initGeometry

- dive down:

GeometrPX

struct

- owns two GlBufferObjects. (construction gets openGL resources)
  - one will be for position and one will be for texture coordinates.
- dive down:

loadSquareGeometry

  - (draw on board, canonical square)
- the data is passed to the VBOs
  - note the binding convention

display

- called by glut when the screen needs updating
- clears screen (you can ignore depth for now)
- dive down

drawSquare

- sets the program (from the SquareShaderState)
- sets some uniform variables in the shaders (more later)
- "hooks up" the VBOs to the appropriate attribute variables
  - makes a GL draw call.
- pop up to display
- swaps: sends to the screen
- checks for errors (which would be printed on the console)
  - note: we could use many different programs and draw lots of things before swapping.

vertex shader

- attribute variables come in
- varying variables go out
- gl_position goes out
  - says where the vertex will go in the window
  - assumes canonical -1..1 square for the display
  - ignore the 3rd and 4th slots for now.

fixed function

- each triangle is rasterized
- at each interior pixel, the varying variables are appropriately blended
- fragment shader is called with this data

fragment shader

- sets fragColor.

- this is a vec4 in RGBA format.

lets play a bit

- lets look at texVbo data which is passed to aTexCoord
- it gets sent on as vTexCoord
- lets use that data for the r and g of the color.

reshape

- called by glut when the window size changes.
- we tell openGL of the new size
- we store this info for our own use
- then we call glutPostRedisplay so that glut will know to trigger a display callback.

lets add a texture

- auxiliary image data
- read from a file, loaded to openGL, used in fragment shader

initTextures

-
- glTexture is a wrapper around a texture handle
- dive:

loadTexture

  - reads from file, turns on any "texture unit", turns on a texture, passes the data.

- binds texture to the GL_TEXTURE_2D target of this unit.
- sets some more magical parameters for the texture.

passing a texture

- to pass textures (see draw square)
- we bind each texture to the GL_TEXTURE_2D target of its own texture unit.
- we send the "texture unit" info as a uniform (see drawSquare)
- in the fragment shader these are of type "sampler2D"

texture coordinates

- we need texture coordinates at each vertex.
  - uses 0-1 unit square
  - we already have this data in a buffer!
  - we will use same texture coords on two textures
- vertex shader just passes this on to a varying variable
- fragment shader makes "texture()" calls.
- returns vec4 in RGBA format.

lets add some interaction

- we will use mouse motion to change the global g_objScale

- this will be sent to the uniform uVertexScale
- this will be used in the vertex shader to change the x coordinate of the vertices
- this will be used in the fragment shader to change the blendings between two texture colors.

interaction

mouse

- callback
- called (due to our registration) whenever the mouse is clicked down or up
- we store this information
  - we need to flip the y coordinate from glut

motion

callback

- called whenever the mouse is moved
- here is where we update g_objScale
- then we call glutPostRedisplay so that glut will know to trigger a display callback.
- see display for use of scale
- see vertex shader for use of scale

keyboard

- s key will create screenshot. (ppm.c)

for the mac

- the mac and glut and openGL 3 are not friends.
- so for mac, we will need to use openGL 2.
- no version numbers in shaders
- in vertex shader in $->$ attribute, out$->$ varying
- in fragment shader in$->$ varying, out$->$ gl_FragColor
- also, no sRGB color space.

your assignment

- get the starter code to run
- improve resizing behavior
- add a triangle to the scene
- use keyboard to move the triangle about