Computer Graphics (CS 543)
Lecture 2 (Part 1): Shader Setup & 2D Graphics Systems

Prof Emmanuel Agu

Computer Science Dept.

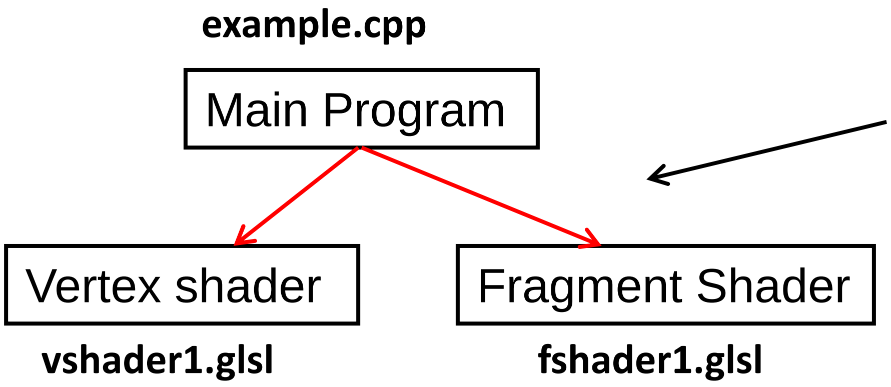
Worcester Polytechnic Institute (WPI)

Recall: OpenGL Program: Shader Setup

- **`initShader( )`**: our homegrown shader initialization
  - Used in main program, connects and link vertex, fragment shaders
  - Shader sources read in, compiled and linked

```
Gluint = program;
```

```
GLuint program = InitShader( "vshader1.glsl", "fshader1.glsl" );
glUseProgram(program);
```



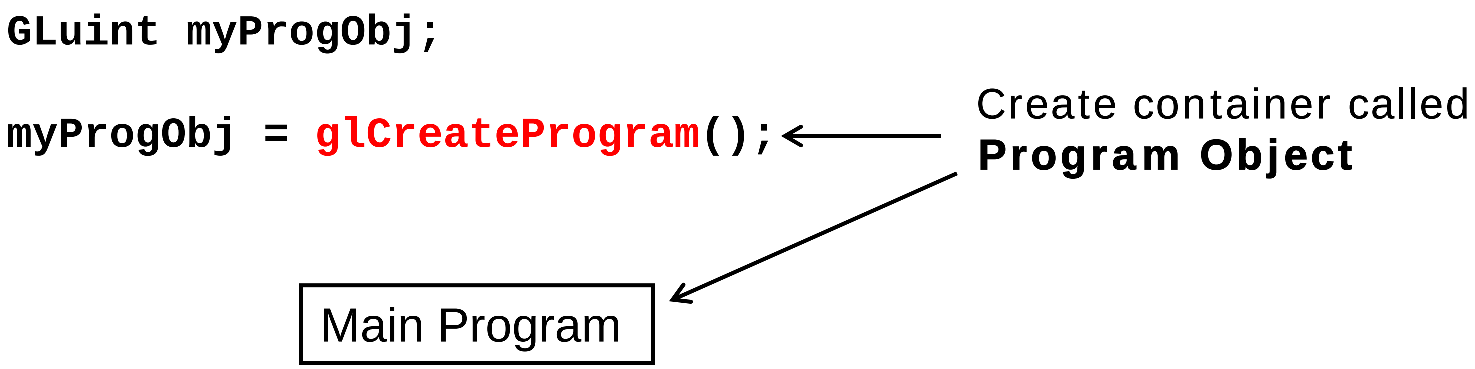
What's inside **initShader??**
Next!

Coupling Shaders to Application

1. Create a program object
2. Read shaders
3. Add + Compile shaders
4. Link program (everything together)
5. Link variables in application with variables in shaders
   - Vertex attributes
   - Uniform variables

Step 1. Create Program Object

- Container for shaders
  - Can contain multiple shaders, other GLSL functions

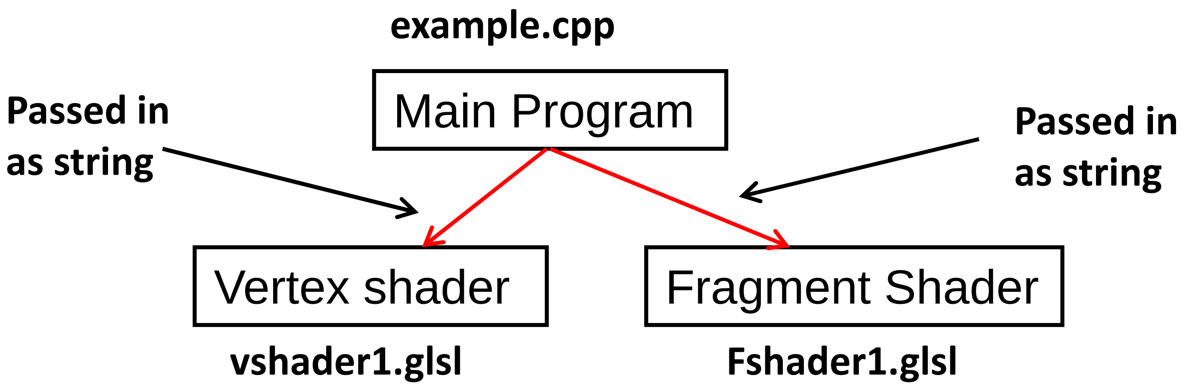


```
GLuint myProgObj;

myProgObj = glCreateProgram();
```

Create container called **Program Object**

Step 2: Read a Shader

- Shaders compiled and added to program object



example.cpp

Main Program

Passed in as string

Passed in as string

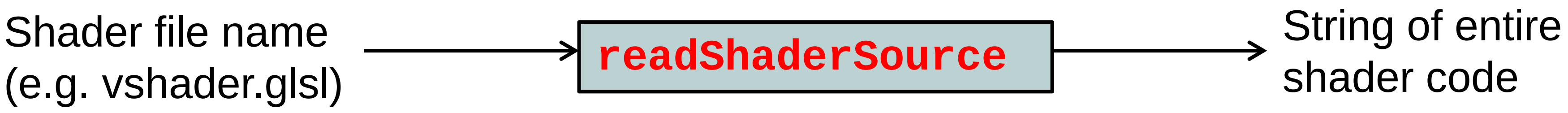
Vertex shader

Fragment Shader

vshader1.glsl

Fshader1.glsl

- Shader file code passed in as null-terminated string using the function **`glShaderSource`**
- Shaders in files (vshader.glsl, fshader.glsl), write function **`readShaderSource`** to convert shader file to string



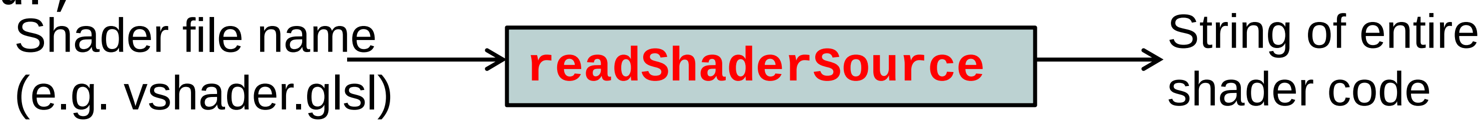
Shader file name (e.g. vshader.glsl)

`readShaderSource`

String of entire shader code

Shader Reader Code?

```
#include <stdio.h>

static char* readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");

    if ( fp == NULL ) { return NULL; }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '\0';
    fclose(fp);

    return buf;
}
```



Shader file name (e.g. vshader.glsl) → readShaderSource → String of entire shader code

Step 3: Adding + Compiling Shaders

```
GLuint myVertexObj;
Gluint myFragmentObj;
```

Declare shader object (container for shader)

```
GLchar vShaderfile[] = “vshader1.glsl”;
Glchar fShaderfile[] = “fshader1.glsl”;
```

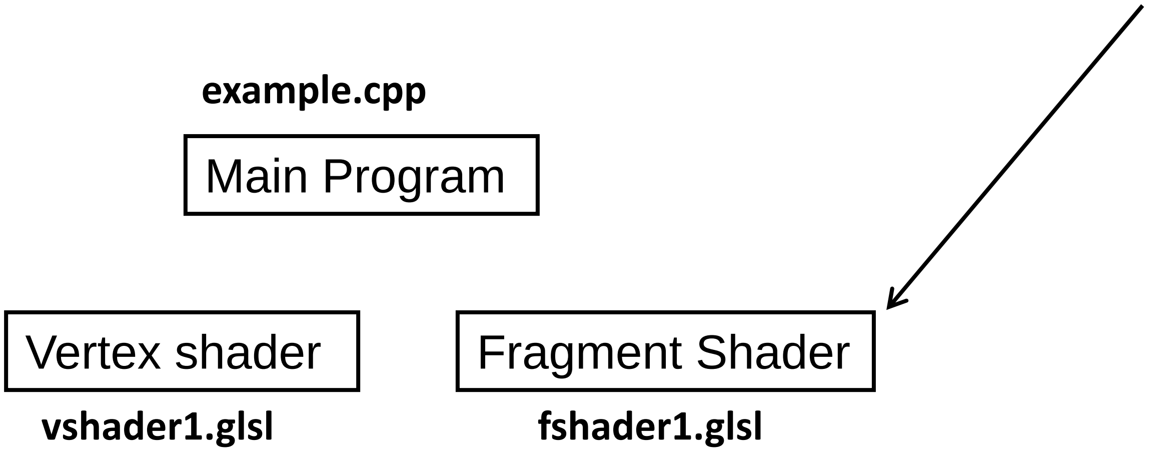
Store names of Shader files

```
GLchar* vSource = readShaderSource(vShaderFile);
GLchar* fSource = readShaderSource(fShaderFile);
```

Read shader files, Convert code to string

```
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
myFragmentObj = glCreateShader(GL_FRAGMENT_SHADER);
```

Create empty Shader objects



Step 3: Adding + Compiling Shaders
Step 4: Link Program

Read shader code strings into shader objects

```
glShaderSource(myVertexObj, 1, vSource, NULL);
glShaderSource(myFragmentObj, 1, fSource, NULL);
```

```
glCompileShader(myVertexObj);
glCompileShader(myFragmentObj);
```

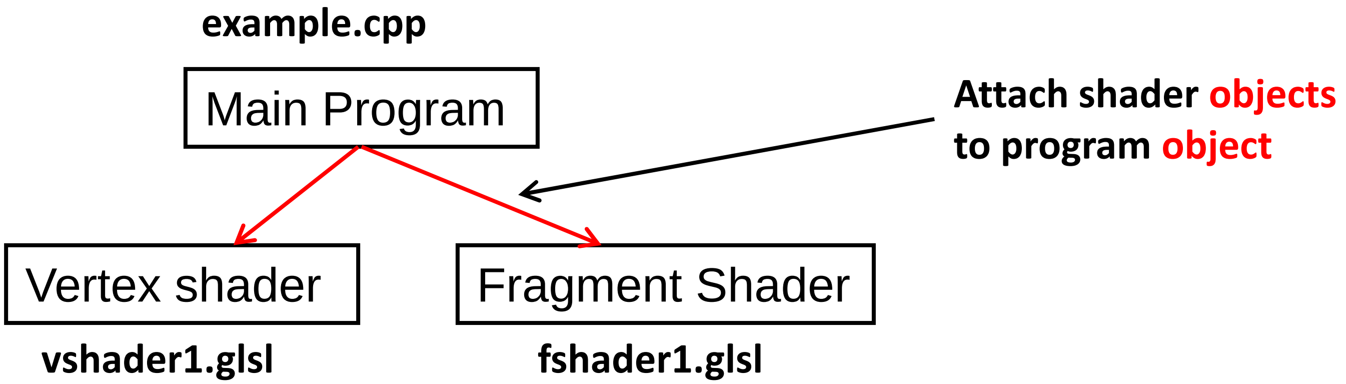
Compile shader objects

```
glAttachShader(myProgObj, myVertexObj);
glAttachShader(myProgObj, myFragmentObj);
```

Attach shader objects
to program object

```
glLinkProgram(myProgObj);
```

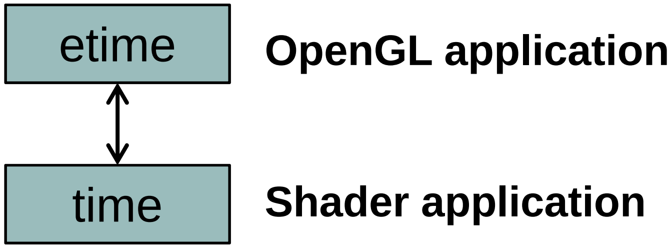
Link Program



Attach shader objects
to program object

Uniform variables

- **Uniform**-qualified variables cannot change = **constants**
- Sometimes want to connect variable in OpenGL application to variable in shader
- Example?
  - Check "elapsed time" variable (etime) in OpenGL application
  - Use elapsed time variable (time) in shader for calculations



Uniform variables

- First declare **etime** variable in OpenGL application, get time

```
 float etime;

etime = 0.001*glutGet(GLUT_ELAPSED_TIME);
```

Elapsed time since program started

- Use corresponding variable **time** in shader

```
 uniform float time;
attribute vec4 vPosition;

main(  ){
      vPosition.x  += (1+sin(time));
     gl_Position = vPosition;
}
```

- Need to connect **etime** in application and **time** in shader!!

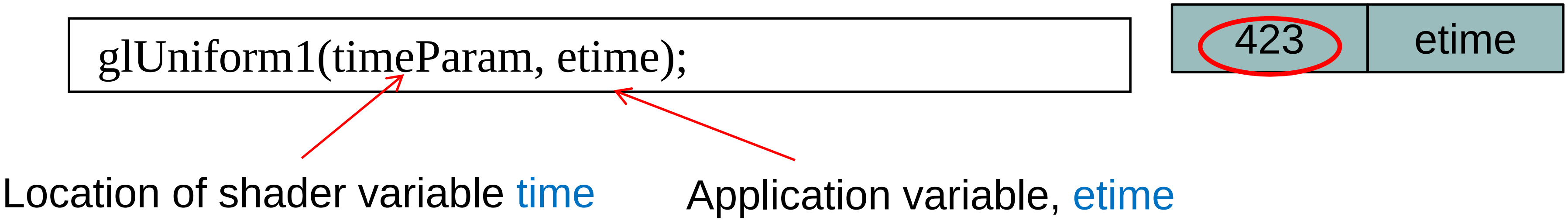
Connecting **etime and time**

- Linker forms table
- Application can get index from table, tie it to application variable
- In application, find location of shader **time** variable in linker table

```
 Glint timeParam;

timeParam = glGetUniformLocation(program, “time”);
```

423	time

- Connect **location** of shader variable **time** location to **etime**!



```
glUniform1(timeParam, etime);
```

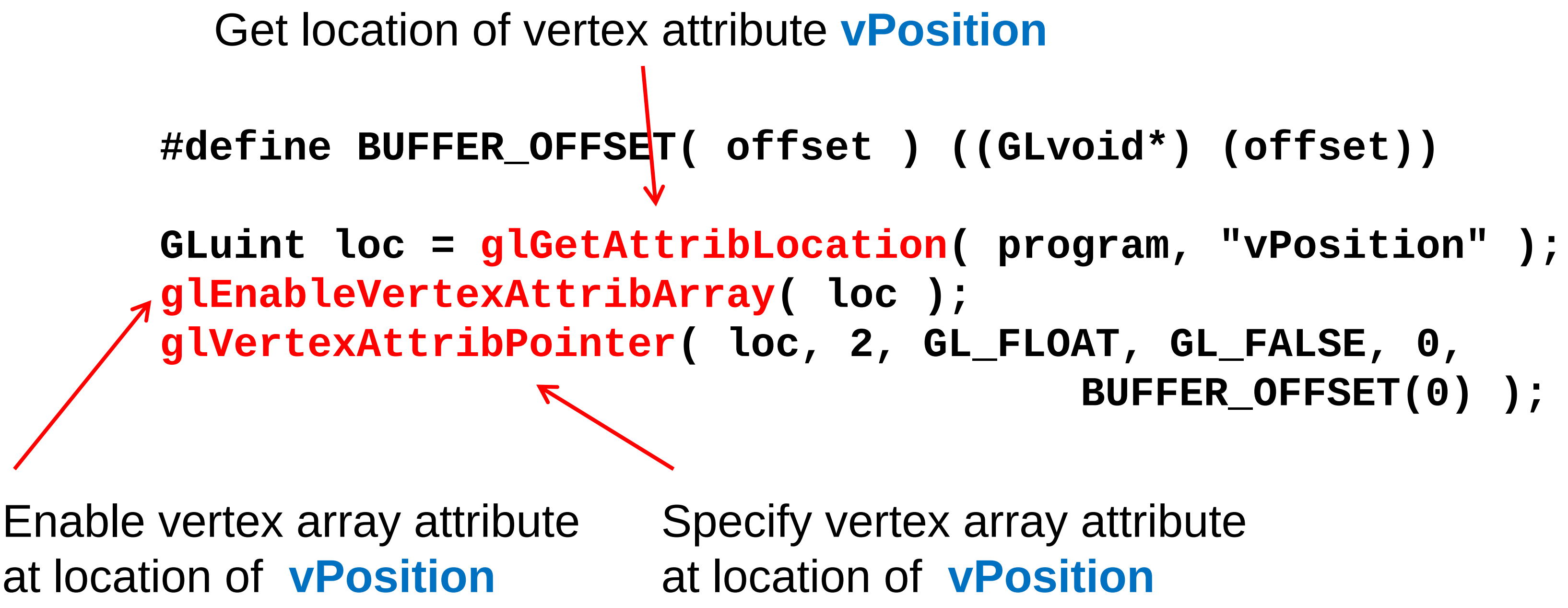
423	etime

Location of shader variable time

Application variable, etime

Vertex Attributes

- Vertex attributes (vertex position, color) are named in the shaders
- Similarly for vertex attributes



Get location of vertex attribute **vPosition**

```
#define BUFFER_OFFSET( offset ) ((GLvoid*) (offset))

GLuint loc = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0,
                                     BUFFER_OFFSET(0) );
```

Enable vertex array attribute
at location of **vPosition**

Specify vertex array attribute
at location of **vPosition**

GLSL

- OpenGL Shading Language
- Vertex and Fragment shaders written in GLSL
- Part of OpenGL 2.0 and up
- High level C-like language
- As of OpenGL 3.1, application must use shaders

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

Example code of vertex shader

Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)

Pointers

- No pointers in GLSL
- Can use C structs that are copied back from functions
- Matrices and vectors are basic types
    - can be passed in and out from GLSL functions
- Example

```
mat3 func(mat3 a)
```

Qualifiers

- GLSL has many C/C++ qualifiers such as **`const`**
- Supports additional ones
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
  - Use **in** qualifier to get to shader
  - **`in float temperature`**
  - **`in vec3 velocity`**

Uniform Qualified

- Variables that are **constant** for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Passing values

- call by **value-return:**
  - Variables copied in
  - Returned values are copied back
- Two possibilities: **in, out**
  - **inout** (deprecated)
- Vertex shader example using **out**

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

Operators and Functions

- Standard C functions
  - Trigonometric: cos, sin, tan, etc
  - Arithmetic: log, min, max, abs, etc
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a;     // a column vector stored as a 1d array
d = a*b;    // a row vector stored as a 1d array
```

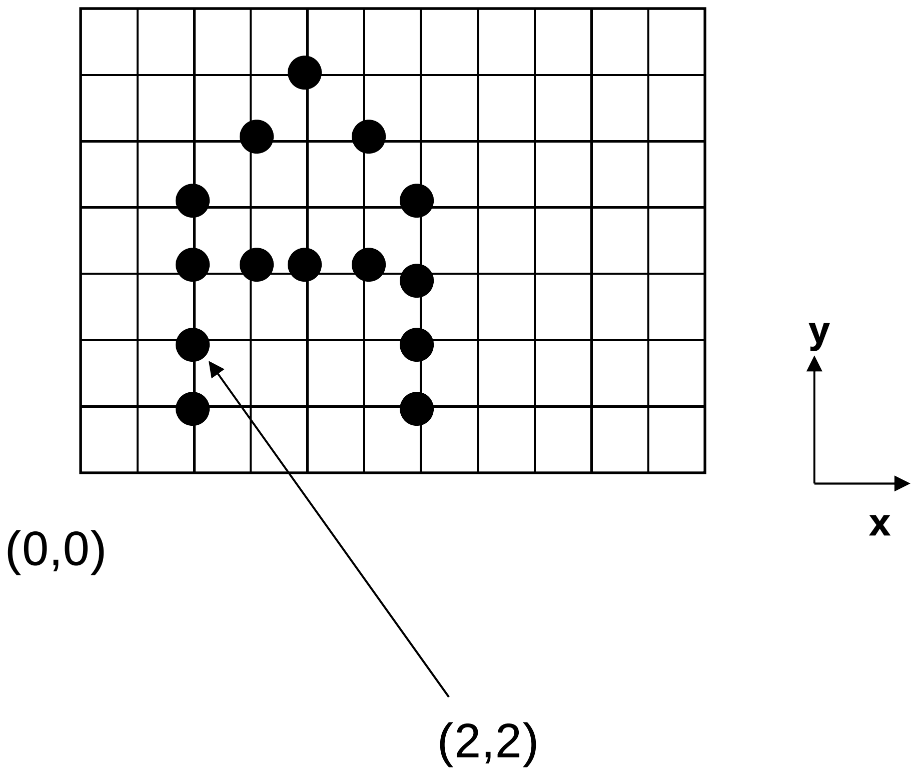
Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **`vec4 a;`**
  - **`a[2], a.b, a.z, a.p`** are the same
- **Swizzling** operator lets us manipulate components

```
a.yz = vec2(1.0, 2.0);
```

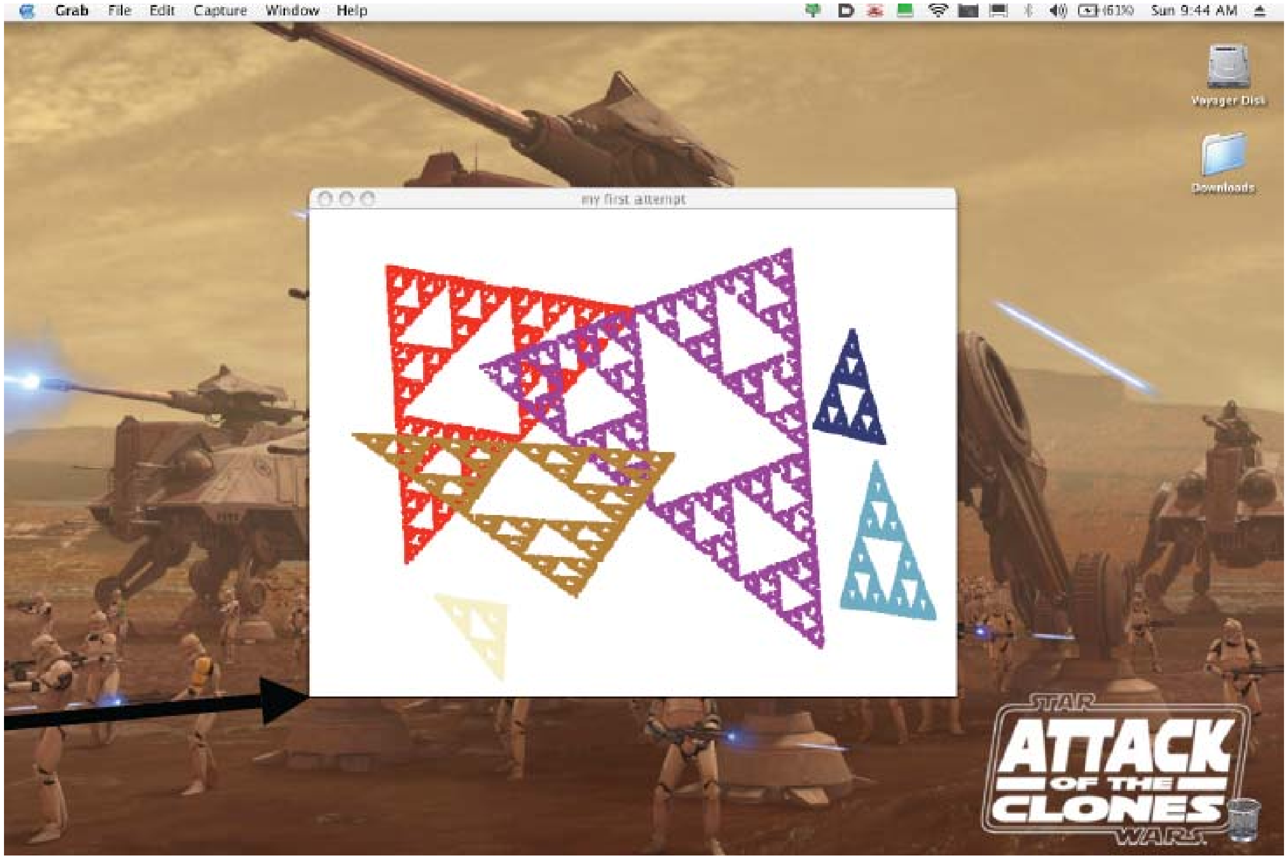
Screen Coordinate System

- Screen: 2D coordinate system (WxH)
- 2D Regular Cartesian Grid
- Origin (0,0): lower left corner (OpenGL convention)
- Horizontal axis – x
- Vertical axis – y
- Pixel positions: grid intersections



Screen Coordinate System

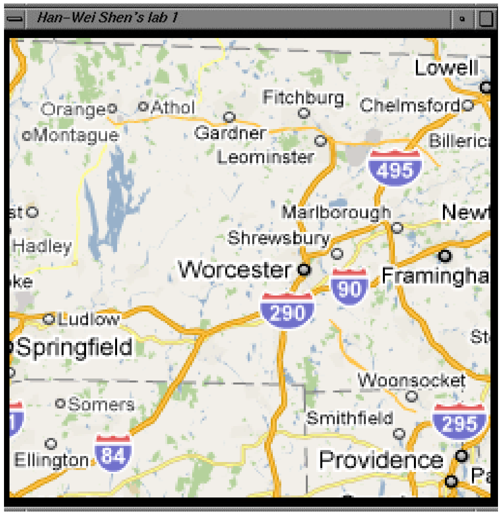
(0,0) is lower left corner of **OpenGL Window.**
NOT lower left corner of entire desktop


Grab File Edit Capture Window Help
(61%) Sun 9:44 AM
Voyager Disk
Downloads
my first attempt
STAR WARS ATTACK OF THE CLONES

OpenGL's (0,0)

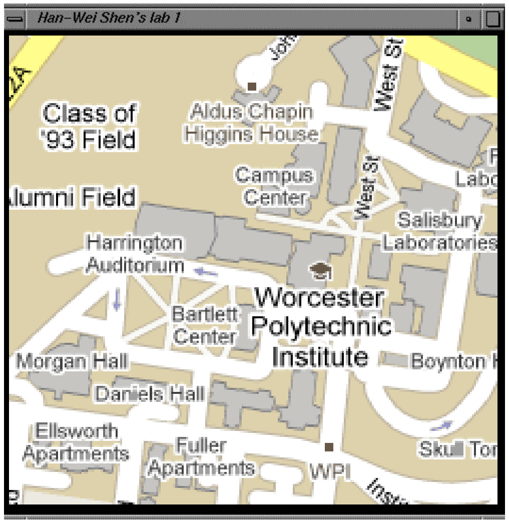
World Coordinate System

- Problems with drawing in screen coordinates:
  - **(x,y) dimensions in pixels:** one mapping, inflexible
  - Not application specific, difficult to use
- World coordinate: application-specific
- E.g: Same screen area. Change input drawing (x,y) range



100 pixels = 30 miles

Change
World window
(mapping)



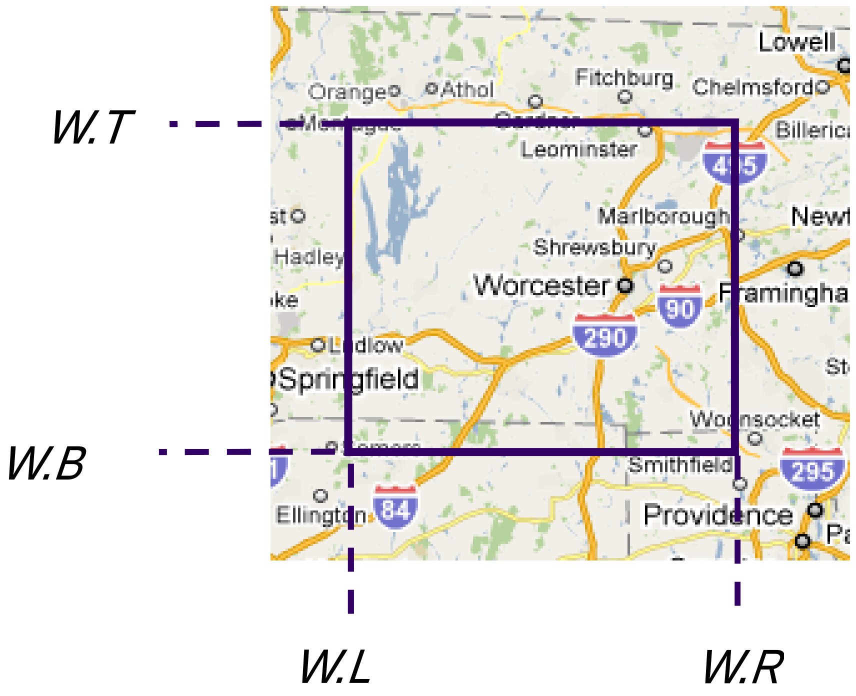
100 pixels = 0.25 miles

Window to Viewport Mapping

- Would like to:
  - Specify drawing in world coordinates (miles, meters, etc)
  - Display in screen coordinates (pixels)
- Need a mapping: ***Window-to-viewport mapping!***
- Basic W-to-V mapping steps:
  1. Define world window
  2. Define viewport
  3. Compute mapping from window to viewport

World Coordinate System

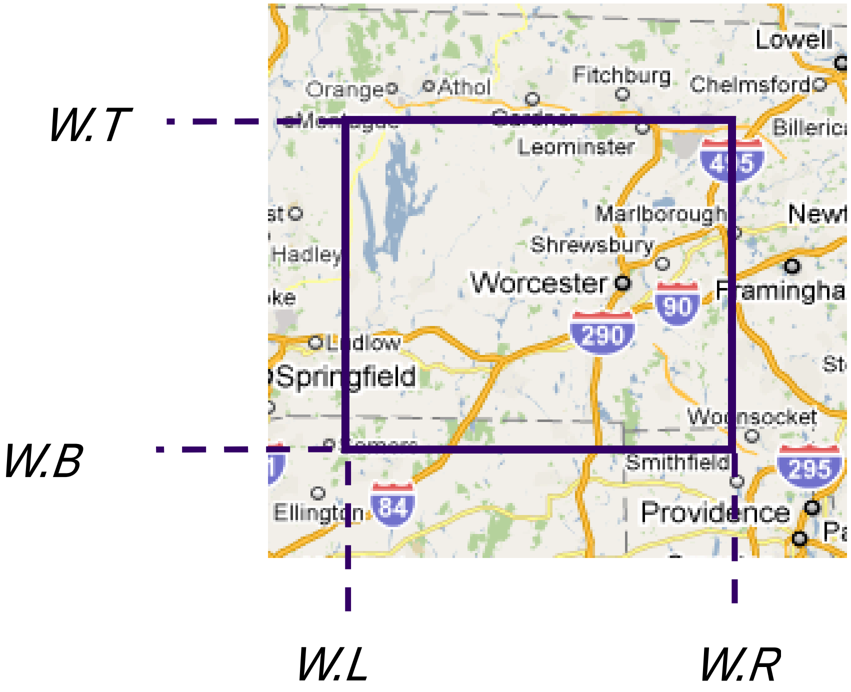
- World Window: region of **source** drawing to be rendered
- Rectangle specified by world window is drawn to screen
- Defined by (left, right, bottom, top) or (*W.L, W.R, W.B, W.T)*



Window to Viewport Mapping

- **Step 1:** Define world window:
  - `Ortho2D(left, right, bottom, top)`

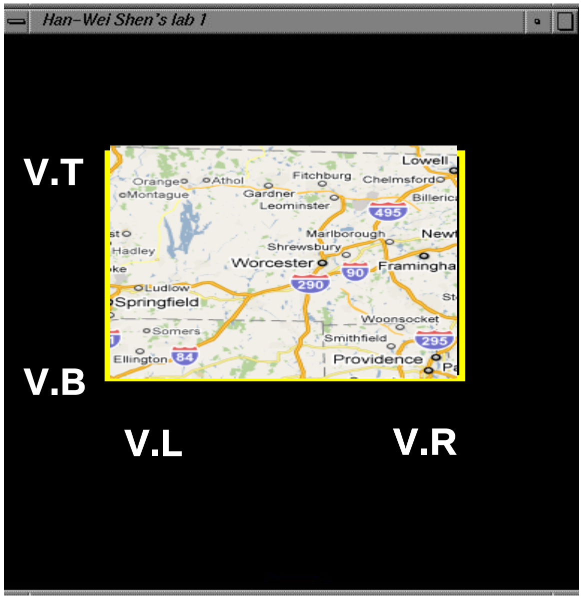
    Or `Ortho2D(W.L, W.R, W.B, W.T)`
  - **Note:** Ortho2D in header file **mat.h**

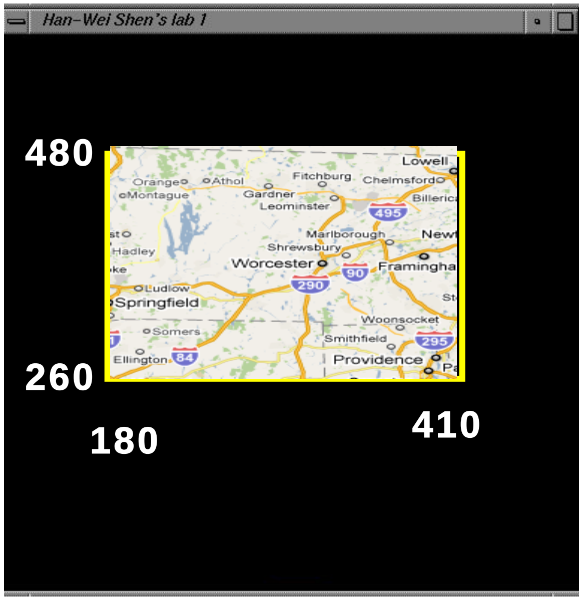


Step 2: Defining a Viewport

- To define viewport

```
glViewport(left, bottom, width, height)
or glViewport(V.L, V.B, V.R – V.L, V.T – V.B)
or glViewport(180, 260, (410 – 180), (480 – 260) )
```





Window to Viewport Mapping

- **Step 3: Draw!**
- Draw as usual with **glDrawArrays**
- All subsequent drawings are automatically mapped

Setting World Window using ortho2D( )

- Include mat.h from book website (Matrix stuff)

```
#include "mat.h"
```

- Declare global to store location in Linker table

```
GLuint ProjLoc;
```

- In OpenGL application (.cpp file) , set viewport

```
    glViewport( 0, 0, w, h );

    .....
```

Setting World Window using ortho2D( )

- Ortho2D (in mat.h) builds matrix for Window Window
- Connect **ortho** matrix to **proj** variable in shader

```
 mat4 ortho = Ortho2D( W.L, W.R, W.B, W.T );

  ProjLoc = glGetUniformLocation( program, "Proj" );
  glUniformMatrix4fv( ProjLoc, 1, GL_FALSE, ortho );
```

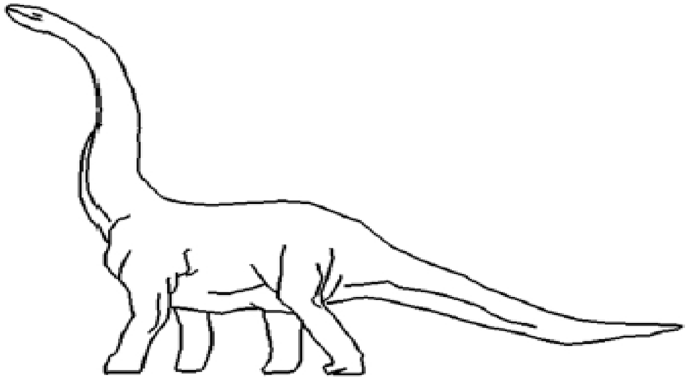
- In shader, multiply each vertex with **proj** matrix

```
uniform mat4 Proj;
in vec4 vPosition;

void main( ){
    gl_Position = Proj * vPosition;
}
```

Drawing Polyline Files

- Problem: want to draw single dino.dat on screen
- Code:

```
    // set world window (left, right, bottom, top)
    Ortho2D(0, 640.0, 0, 440.0);

  //  now set viewport (left, bottom, width, height)
     glViewport(0, 0, 64, 44);

 // Draw polyline fine
     drawPolylineFile(dino.dat);
```



References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 2
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 2