CS380: Introduction to Computer Graphics
Introduction to OpenGLSL

Min H. Kim
KAIST School of Computing

INTRODUCTION TO OPENGL

References:
https://www.khronos.org/opengl/wiki/History_of_OpenGL
https://www.cg.tuwien.ac.at/courses/CG23/slides/tutorials/CG2LU_OpenGL_3.x_Introduction_pt_1.pdf

What is OpenGL?

- OpenGL = Open Graphics Library
- An open industry-standard API for hardware accelerated graphics drawing
- Implemented by graphics-card vendors
- Maintained by the Khronos-Group

What is OpenGL?

- Pros & Cons:
  - + Full specification freely available
  - + Everyone can use it
  - + Can use it anywhere (Windows, Linux, Mac, BSD, Mobile phones, Web-pages (soon), ...)
  - + Long-term maintenance for older applications
  - + New functionality usually earlier available through Extensions
  - - Inclusion of Extensions to core may take longer
  - ? Game-Industry

Setup OpenGL Project

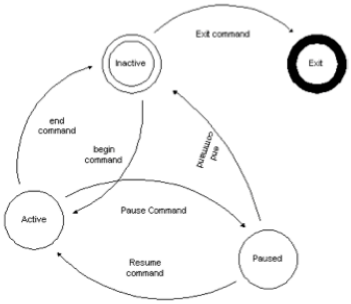
- Include OpenGL-header:
  #include <GL/gl.h> // basic OpenGL
- Link OpenGL-library “opengl32.lib”
- If needed, also link other libraries (esp. GLEW, see later!).

OpenGL in more detail

- OpenGL-functions prefixed with “gl”:
  glFunction{1234}{bsifd...}{v}(T arg1, T arg2, ...);
  Example: glDrawArrays(GL_TRIANGLES, 0, vertexCount);
- OpenGL-constants prefixed with “GL_”:
  GL_SOME_CONSTANT
  Example: GL_TRIANGLES
- OpenGL-types prefixed with “GL”:
  GLtype
  Example: GLfloat

OpenGL in more detail

- OpenGL is a state-machine
- Remember state-machines:
  - Once a state is set, it remains active until the state is changed to something else via a transition.
  - A transition in OpenGL equals a function-call.
  - A state in OpenGL is defined by the OpenGL-objects which are current.



OpenGL in more detail

- Set OpenGL-states:

```
glEnable(...);
glDisable(...);
gl*(...); // several call depending on purpose
```

- Query OpenGL-states with Get-Methods:

```
glGet*(...); // several calls available, depending on
what to query
```

OpenGL 2.1

- Released in August 2006
- Fully supported “fixed function” (FF)
- GLSL-Shaders supported as well
- Mix of FF and shaders was possible, which could get confusing or clumsy quickly in bigger applications
- Supported by all graphics-drivers

OpenGL 3.0

- Released in August 2008
- Introduced a deprecation model:
  - Mainly FF was marked deprecated
  - Use of FF still possible, but not recommend
- Also introduced Contexts:
  - Forward-Compatible Context (FWD-CC) vs. Full Context
- With FWD-CC, no access to FF anymore, i.e. FF-function-calls create error “Invalid Call”

OpenGL 3.0 (cont.)

- Furthermore, GLSL 1.3 was introduced
- Supported by recent Nvidia and ATI-graphics drivers.

OpenGL 3.1

- Released in March 2009
- Introduced GLSL 1.4
- Removed deprecated features of 3.0, but FF can still be accessed by using the "GL_ARB_compatibility"-extension.
- Supported by recent Nvidia and ATI-graphics drivers.

OpenGL 3.2

- Released in August 2009
- Profiles were introduced: Core-Profile vs. Compatibility-Profile
- With Core-Profile, only access to OpenGL 3.2 core-functions
- With Compatibility-Profile, FF can still be used
- Also introduced GLSL 1.5
- Supported by recent Nvidia and ATI-graphics drivers.

OpenGL 3.3

- Released on 10th March 2010
- Introduces GLSL 3.3
- Includes some new Extensions
- Maintains compatibility with older hardware

OpenGL 4.0

- Released on 10th March 2010
- Introduces GLSL 4.0
- Introduces new shader-stages for hardware-tessellation (tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages.)
- Adoption of new Extensions to Core.

OpenGL 4.6

- Released on July 30, 2017
- Supports GLSL 4.6
- State-of-the-art OpenGL version
- Refer to the Khronos website for more information: https://www.khronos.org/opengl/wiki/History_of_OpenGL

GLEW

- On Windows only OpenGL 1.1 supported natively.
- To use newer OpenGL versions, each additional function, i.e., all extensions (currently ~1900), must be loaded manually!
- →Lots of work! Therefore:
- Use GLEW = OpenGL Extension Wrangler

GLEW

- Include it in your program and initialize it:

```
#include <GL/glew.h> // include before other GL headers!
// #include <GL/gl.h> included with GLEW already

void initGLEW() {
    GLenum err = glewInit(); // initialize GLEW
    if (err != GLEW_OK) // check for error {
        cout << "GLEW Error: " << glewGetErrorString(err);
        exit(1);
    }
}
```

GLEW

- Check for supported OpenGL version:

```
if (glewIsSupported("GL_VERSION_3_2")) {
  // OpenGL 3.2 supported on this system
}
```

- To check for a specific extension:

```
if (GLEW_ARB_geometry_shader4) {
  // Geometry-Shader supported on this system
}
```


GLEW

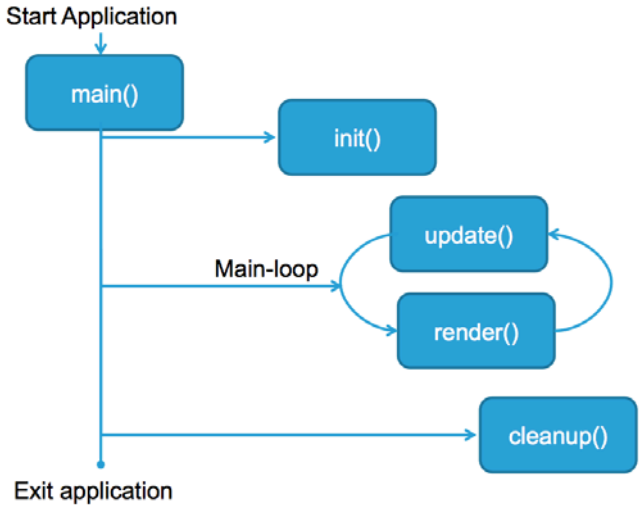
- How to use:
  - Include GLEW-header:

```
#include <GL/glew.h> // GLEW
```

  - Link OpenGL-library “opengl32.lib” and “glew32.lib”
  - Copy “glew32.dll” to bin folder
  - You’re ready to go.

OpenGL Program Skeleton

- Typical OpenGL-Application:



OpenGL Program Skeleton

- main():
  - Program-Entry
  - Create window
  - Call init()
  - Start main window-loop Call cleanup()
  - Exit application
- init():
  - Initialize libraries, load config-files, ...
  - Allocate resources, preprocessing, ...

OpenGL Program Skeleton

- update():
  - Handle user-input, update game-logic, ...
- render():
  - Do actual rendering of graphics here!
  - Note: Calling render() twice without calling update() in between should result in the same rendered image!
- cleanup():
  - Free all resources

OpenGL Program Skeleton

- Example init()-function:

```
void init() {
        // Create and initialize a window with depth-buffer and
        // double- buffering. See your window-managers documentation.

        // enable the depth-buffer in OpenGL
        glEnable(GL_DEPTH_TEST);

        // enable back-face culling in OpenGL
        glEnable(GL_CULL_FACE);

        // define a clear color
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        // set the OpenGL-viewport
        glViewPort(0, 0, windowWidth, windowHeight);

        // Do other useful things
}
```

OpenGL Program Skeleton

- The geometry of a 3D-object is stored in an array of vertices called Vertex-Array.
- Each vertex can have so called Attributes, like a Normal Vector and Texture-Coordinates.
- OpenGL also treats vertices as attributes!
- To render geometry in OpenGL, vertex-(attribute)-arrays are passed to OpenGL and then rendered.

OpenGL Program Skeleton

- To do so:
  - Query the attribute-location in the shader:

```
GLint vertexLocation = glGetAttribLocation(
        myShaderProgram, "in_Position”);
```

  - Enable an array for the vertex-attribute:

```
glEnableVertexAttribArray(vertexLocation);
```

  - Then tell OpenGL which data to use:

```
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
GL_FALSE, 0, myVertexArray);
```

OpenGL Program Skeleton

- Draw (“render”) the arrays:

  glDrawArrays(GL_TRIANGLES, 0, 3); // this does the actual draw

- Finally disable the attribute-array:

  glDisableVertexAttribArray(vertexLocation);

OpenGL Program Skeleton

- Example render()-function:

```
// triangle data
static GLfloat vertices[] = { -0.5, -0.333333, 0, // x1, y1, z1
                              +0.5, -0.333333, 0, // x2, y2, z2
                              +0.0, +0.666666, 0}; // x3, y3, z3
...
void render() {
    // clear the color-buffer and the depth-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // activate a shader program
    glUseProgram(myShaderProgram);

    // Find the attributes
    GLint vertexLocation = glGetAttribLocation(
        myShaderProgram, "in_Position");
```

OpenGL Program Skeleton

```
        // enable vertex attribute array for this attribute
        glEnableVertexAttribArray(vertexLocation);

        // set attribute pointer
        glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
              GL_FALSE, 0, vertices);

        // Draw ("render") the triangle
        glDrawArrays(GL_TRIANGLES, 0, 3);

        // Done with rendering. Disable vertex attribute array
        glDisableVertexAttribArray(vertexLocation);

        // disable shader program
        glUseProgram(0);

        // Swap buffers
}
```


OpenGL-Object life-cycle

- In OpenGL, all objects, like buffers and textures, are somehow treated the same way.
- On object creation and initialization:
  - First, create a handle to the object (in OpenGL often called a name). Do this ONCE for each object.
  - Then, bind the object to make it current.
  - Pass data to OpenGL. As long as the data does not change, you only have to do this ONCE.
  - Unbind the object if not used.

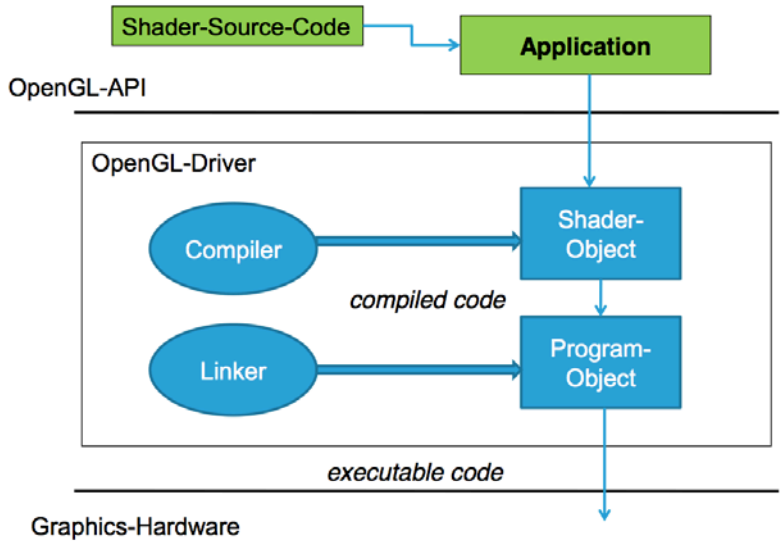
OpenGL-Object life-cycle

- On rendering, or whenever the object is used:
  - Bind it to make it current.
  - Use it.
  - Unbind it.
- Finally, when object is not needed anymore:
  - Delete object.
  - Note that in some cases you manually have to delete attached resources!
- NOTE: OpenGL-objects are NOT objects in an OOP-sense!

What shaders are

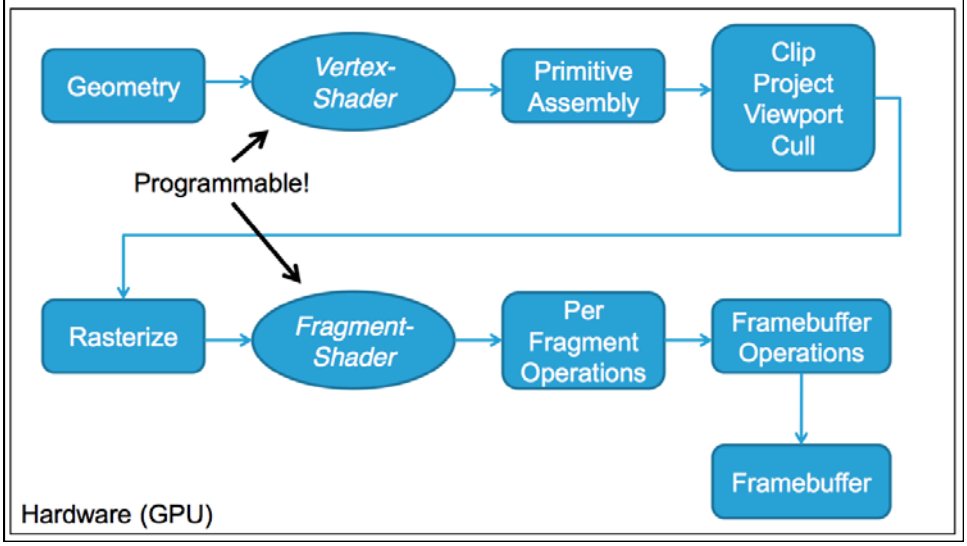
- Small C-like programs executed on the graphics-hardware
- Replace fixed function pipeline with shaders
- Shader-Types
  - Vertex Shader (VS): per vertex operations
  - Geometry Shader (GS): per primitive operations
  - Fragment Shader (FS): per fragment (pixel) operations, so-called Pixel Shader
- Used e.g. for transformations and lighting

Shader-Execution model



Rendering-Pipeline

- OpenGL 3.x Rendering-Pipeline:



Rendering-Pipeline

- Remember:
  - The Vertex-Shader is executed ONCE per each vertex!
  - The Fragment-Shader is executed ONCE per rasterized fragment (~ a pixel)!
- A Shader-Program consists of both,
  - One VS
  - One FS

Setting up shaders and programs

- Compile shaders:

```
char* shaderSource; // contains shadersource
int shaderHandle = glCreateShader(GL_SHADER_TYPE);

// shader-types: vertex || geometry || fragment
glShaderSource(shaderHandle, 1, shaderSource, NULL);
glCompileShader(shaderHandle);
```

- Create program and attach shaders to it:

```
int programHandle = glCreateProgram();
glAttachShader(programHandle, shaderHandle);
// do this for vertex AND fragment-shader (AND geometry if needed)!
```

- Finally link program:

```
glLinkProgram(programHandle);
```

Enabling shaders

- Enable a GLSL program:

  glUseProgram(programHandle); // shader-program now active

  – The active shader-program will be used until glUseProgram() is called again with another program-handle.

  – Call of glUseProgram(0) sets no program active (undefined state!).

Basic shader layout

- Shader-Programs must have a main()-method
- Vertex-Shader outputs to at least gl_Position
- Fragment-Shader to custom defined output

```
//preprocessor directives like:
#version 150

//variable declarations

void main() {
    //do something and write into output variables
}
```

Shader Parameter

- Shader variable examples:

```
uniform mat4 projMatrix; // uniform input
in vec4 vertex; // attribut-input
out vec3 fragColor; // shader output
```

- Three types:
  - uniform: does not change per primitive; read-only in shaders
  - in: VS: input changes per vertex, read-only;
    FS: interpolated input; read-only
  - out: shader-output; VS to FS; FS output.

Uniform Parameter

- Set uniform parameters in an application:

```
// first get location
projMtxLoc = glGetUniformLocation(programHandle,
  "projMatrix");

// then set current value
glUniformMatrix4fv(projMtxLoc, 1, GL_FALSE,
currentProjectionMatrix);
```

  - First get the „location“ of the uniform-variable
  - Then set the current value
  - Can pass values to vertex- and fragment-shader

Attribute Parameter

- A vertex can have attributes like a normal-vector or texture-coordinates
- OpenGL also treats the vertex itself as an attribute
- We want to access our current vertex within our vertex-shader (as we used to do with gl_Vertex in former GLSL-versions):
- Therefore, we declare in our vertex-shader:

```
in vec4 vertex; // vertex attribut
```


Attribute Parameter

- Now, there are two ways to pass data to this shader attribute-variable, depending on:
  - if you just have an array of vertices (Vertex Array),
  - or an VBO (Vertex Buffer Object).
- To do so: Query shader-variable location
  - Enable vertex-attribute array
  - Set pointer to array
  - Draw and disable array

Attribute Parameter: Vertex-Array

- For a Vertex-Array, pass data like this:

```
// first get the attribute-location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// enable an array for the attribute
glEnableVertexAttribArray(vertexLocation);

// set attribute pointer
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0,
myVertexArray);

// Draw ("render") the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

// Done with rendering. Disable vertex attribute array
glDisableVertexAttribArray(vertexLocation);
```


Attribute Parameter: VBO

- Setting attribute parameters with VBO:

```
// first get location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// activate desired VBO
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

// set attribute-pointer
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE,
0, 0);

// finally enable attribute-array
glEnableVertexAttribArray(vertexLocation); ...
```

Fragment Output

- Since GLSL 1.3, gl_FragColor is deprecated.
- Therefore, need to define output on our own.
- Declare output variable in FS:

```
out vec4 fragColor; // fragment color output
```

- In the application, before linking the shader-program with glLinkProgram(), bind the FS-output:

```
glBindFragDataLocation(programHandle, 0,
"fragColor");
```

- Finally assign a value to fragColor in the FS.

Example usage

- An application using shaders could basically look like this:

```
//Load shader and initialize parameter-handles

//Do some useful stuff like binding texture, activate
//texture-units, calculate and update matrices, etc.

//glUseProgram(programHandle);

//Set shader-parameters
//Draw geometry

//glUseProgram(anotherProgramHandle);
...
```

Appendix A

HELLO WORLD 2D

Code (hw2d.zip)

- we use c++
  – see 'C++ primer'.
- now we look at the code from the textbook
- Core codes:
  – asst1.cpp
  – glsupport.h/.cpp
  – asst1-gl3.fshader
  – asst1-gl3.vshader
- Auxiliary codes:
  – ppm.h/.cpp

Code (hw2d.zip)

asst1.cpp

- Set g_Gl2Compatible = true to use GLSL 1.0 and g_Gl2Compatible = false to use GLSL 1.3.
- on Mac OS X currently there is no way of using OpenGL 3.x with GLSL 1.3 when GLUT is used.
- If g_Gl2Compatible=true, shaders with -gl2 suffix will be loaded.
- If g_Gl2Compatible=false, shaders with -gl3 suffix will be loaded.
- To complete the assignment you only need to edit the shader files that get loaded

asst1.cpp

- How to check the version you have

```
cout << "GL ver: " << glGetString(GL_VERSION) <<"\n";
cout << "GLEW ver: " << glewGetString(GLEW_VERSION) <<"\n";
cout << "GLSL ver: "<<
glGetString(GL_SHADING_LANGUAGE_VERSION) <<"\n";
```

- Outputs

```
GL ver: 2.1 NVIDIA-8.12.47 310.40.00.05f01
GLEW ver: 1.10.0
GLSL ver: 1.20
```


asst1.cpp

```
int main(int argc, char **argv) {
 try {
      initGlutState(argc,argv);
      glewInit(); // load the OpenGL extensions

      initGLState();
      initShaders();
      initGeometry();
      initTextures();

      glutMainLoop();
      return 0;
 }
 catch (const runtime_error& e) {
      cout << "Exception caught: " << e.what() << endl;
      return -1;
 }
}
```

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

- needs include and library files, and installed drivers.
- 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
- Our current codes on Mac:
  - OpenGL 3.2 (GLSL 1.3) doesn't work
  - OpenGL 2.1 (GLSL 1.2) works instead

GLUT

- library of functions to talk with the windowing system
- open up windows
- glut can inform you when some “event” occurs – mousemove, buttonpress, windowresize
- 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 (windows/X)

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 (SIMD)
- 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
    - future support unclear.

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

initGlutState()

- 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

initGlutState()

```
static void initGlutState(int argc, char **argv) {
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
    glutInitWindowSize(g_width, g_height);     // create a window
    glutCreateWindow("CS492B: Hello World");   // title the window

    glutDisplayFunc(display);              // display rendering callback
    glutReshapeFunc(reshape);                // window reshape callback
    glutMotionFunc(motion);                // mouse movement callback
    glutMouseFunc(mouse);                  // mouse click callback
    glutKeyboardFunc(keyboard);
}
static void reshape(int w, int h) {
 g_width = w;
 g_height = h;
 glViewport(0, 0, w, h);
 glutPostRedisplay();
}
```


Double buffering

- The monitor displays one image at a time
- So if we render the next image to screen, then rendered primitives pop up
- And even worse, we often clear the screen before generating a new image
- A better solution is ”double buffering”

Double buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

initGLState()

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

```
static void initGLState() {
    glClearColor(128./255,200./255,1,0);
    glEnable(GL_FRAMEBUFFER_SRGB);
}
```

Vertex Buffer Object (VBO)

```
static GLfloat sqVerts[12] = {
    -.5, -.5,
    .5,  .5,
    .5,  -.5,

    -.5, -.5,
    -.5, .5,
    .5,  .5
  };

static GLfloat sqCol[18] =  {
    1, 0, 0,
    0, 1, 1,
    0, 0, 1,

    1, 0, 0,
    0, 1, 0,
    0, 1, 1
  };
```


initVBOs()

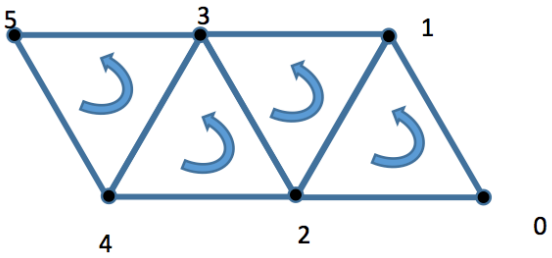
- Transmitting the vertex data from the CPU to the GPU is an expensive process!
- Vertex Buffer Object is to transfer vertex data over to OpenGL.
- in our case, we have attributes for position, color, (and later, texture coordinates)
- when the square geometry is created, the data is passed to the VBOs

initVBOs()

```
static void initVBOs(void) {
    glGenBuffers(1, &sqVertB0); // create a vertex butter
    glBindBuffer(GL_ARRAY_BUFFER, sqVertB0); // bind the buffer to the GL buffer
    glBufferData( // pass data to the GL buffer
        GL_ARRAY_BUFFER,
        12*sizeof(GLfloat),
        sqVerts,
        GL_STATIC_DRAW);

    glGenBuffers(1, &sqColB0)
    glBindBuffer(GL_ARRAY_BUFFER, sqColB0);
    glBufferData(
        GL_ARRAY_BUFFER,
        18*sizeof(GLfloat),
        sqCol,
        GL_STATIC_DRAW);
}
```


initVBOs()

- OpenGL Order: (0, 1, 2), (2, 1, 3), (2, 3, 4), ...
- Consist orientation of the triangles generated.



Display

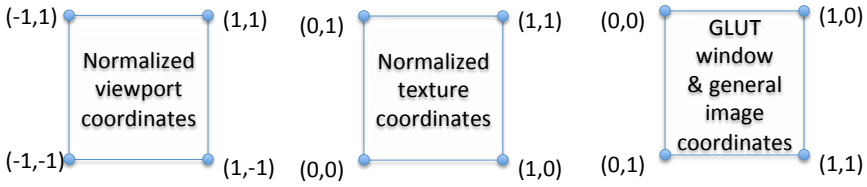
- called by glut when the screen needs updating
- clears screen (you can ignore depth for now)
- sets the program (from the shaderState)
- sets some uniform variables in the shaders (more later)
- draws the geometry

Display()

```
static void display(void) {
        glUseProgram(h_program)
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        drawObj(sqVertB0, sqColB0, 6);
        glutSwapBuffers();
}
void drawObj(const ShaderState& curSS) {
        glBindBuffer(GL_ARRAY_BUFFER, vertbo);
        safe_glVertexAttribPointer2(h_aVertex);
        safe_glEnableVertexAttribArray(h_aVertex);

        glBindBuffer(GL_ARRAY_BUFFER, colbo);
        safe_glVertexAttribPointer3(h_aColor);
        safe_glEnableVertexAttribArray (h_aColor);

        glDrawArrays(GL_TRIANGLES,0,numverts);

        safe_glDisableVertexAttribArray(h_aVertex);
        safe_glDisableVertexAttribArray(h_aColor);
}
```

Vertex Shader

- Typical use of the v-shader:
  - to determine the final position of the vertices on the screen
- Input: the attribute variables (position, color, text. coor.)
- Output: the reserved output variable, gl_Position
- The x and y coordinates of this variable are interpolated as positions within the drawing window.



Vertex Shader

```
#version 130

uniform float uVertexScale;

in vec2 aPosition;
in vec3 aColor;
in vec2 aTexCoord0, aTexCoord1;

out vec3 vColor;
out vec2 vTexCoord0, vTexCoord1;

void main() {
  gl_Position = vec4(aPosition.x * uVertexScale, aPosition.y, 0,1);
  vColor = aColor;
  vTexCoord0 = aTexCoord0;
  vTexCoord1 = aTexCoord1;
}
```

Fragment Shader

- After rasterization (it's not programmable)
- Input: interpolated data of varying variables
- Output: color values in the framebuffer (a part of GPU memory)
- Control the material and geometric properties of the material at that pixel

Fragment Shader

```
#version 130

uniform float uVertexScale;
uniform sampler2D uTexUnit0, uTexUnit1;

in vec2 vTexCoord0, vTexCoord1;
in vec3 vColor;

out vec4 fragColor;

void main(void) {
  vec4 color = vec4(vColor.x, vColor.y, vColor.z, 1);
  vec4 texColor0 = texture(uTexUnit0, vTexCoord0);
  vec4 texColor1 = texture(uTexUnit1, vTexCoord1);

  float lerper = clamp(.5 *uVertexScale, 0., 1.);
  float lerper2 = clamp(.5 * uVertexScale + 1.0, 0.0, 1.0);

  fragColor = ((lerper)*texColor1 + (1.0-lerper)*texColor0) * lerper2 + color * (1.0-lerper2);
}
```

As varying variables, the texture coordinates are interpolated from their vertex values to appropriate values at each pixel.

Texture

- auxiliary image data
- read from a file, loaded to openGL, used in fragment shader
- initTextures
- glTexture is a wrapper around a texture handle
- loadTexture
- – reads from file, turns on any “texture unit”, turns on a texture, passes the data.
- binds each texture to a unit (we have 2 here)
- we will send the “texture unit” info as a uniform (see display)

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.

Interaction

- we will use mouse motion to change the global g object
- this will be sent to the uniform uVertexScale
- this will be used in the vertex shader to change the x coordinate of the vertices

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

Interaction: Motion

- callback
- called whenever the mouse is moved
- here is where we update g objectScale
- 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