
Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Menus

- Adding menu that pops up on mouse click

1. Create menu using `glutCreateMenu(myMenu);`

2. Use `glutAddMenuEntry` adds entries to menu

3. Attach menu to mouse button (left, right, middle) using `glutAttachMenu`
Menus

- Example:

```c
void mymenu(int value){
    if(value == 1){
        glClear(GL_COLOR_BUFFER_BIT);
        glFlush();
    }
    if (value == 2) exit(0);
}
```

shows on menu

checked in mymenu

```c
... glutCreateMenu(myMenu);
    glutAddMenuEntry("Clear Screen", 1);
    glutAddMenuEntry("Exit", 2);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
... 
```
GLUT Interaction using other input devices

- Tablet functions (mouse cursor must be in display window)

```c
glutTabletButton (tabletFcn);
```

```c
void tabletFcn(Glint tabletButton, Glint action, Glint xTablet, Glint yTablet)
```

- Spaceball functions
- Dial functions
- Picking functions: use your finger
- Menu functions: minimal pop-up windows within your drawing window
- Reference: *Hearn and Baker, 3rd edition (section 20-6)*
Adding Interaction

- So far, OpenGL programs just render images
- Can add user interaction
- Examples:
  - User hits ‘h’ on keyboard -> Program draws house
  - User clicks mouse left button -> Program draws table
Types of Input Devices

- **String**: produces string of characters e.g. keyboard
- **Locator**: User points to position on display. E.g. mouse
Types of Input Devices

- **Valuator:** generates number between 0 and 1.0
- **Pick:** User selects location on screen (e.g. touch screen in restaurant, ATM)
Using Keyboard Callback for Interaction

1. register callback in main() function
   
   ```c
   glutKeyboardFunc( myKeyboard );
   ```

2. implement keyboard function
   
   ```c
   void myKeyboard(char key, int x, int y )
   {
      // put keyboard stuff here
      ..........
      switch(key){
         // check which key
         case 'f':
            // do stuff
            break;
         case 'k':
            // do other stuff
            break;
      }
   }
   ```

   Note: Backspace, delete, escape keys checked using their ASCII codes
Keyboard Interaction

- For function, arrow and other special-purpose keys, use

```c
glutSpecialFunc (specialKeyFcn);
...
Void specialKeyFcn (Glint specialKey, GLint, xMouse, GLint yMouse)
```

- Example: if (specialKey == GLUT_KEY_F1) // F1 key pressed
  - GLUT_KEY_F1, GLUT_KEY_F12, .... for function keys
  - GLUT_KEY_UP, GLUT_KEY_RIGHT, .... for arrow keys
  - GLUT_KEY_PAGE_DOWN, GLUT_KEY_HOME, .... for page up, home keys

- Complete list of special keys designated in `glut.h`
Mouse Interaction

- Declare prototype
  - `myMouse(int button, int state, int x, int y)`
  - `myMovedMouse`

- Register callbacks:
  - `glutMouseFunc(myMouse)`: mouse button pressed
  - `glutMotionFunc(myMovedMouse)`: mouse moves with button pressed
  - `glutPassiveMotionFunc(myMovedMouse)`: mouse moves with no buttons pressed

- Button returned values:
  - `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`

- State returned values:
  - `GLUT_UP`, `GLUT_DOWN`

- X,Y returned values:
  - x,y coordinates of mouse location
Mouse Interaction Example

- Each mouse click generates separate events
- Store click points in **global** or **static** variable in mouse function
- **Example**: draw (or select) rectangle on screen
- Mouse y returned assumes y=0 at top of window
- OpenGL assumes y=0 at bottom of window. Solution? Flip mouse y

```c
void myMouse(int button, int state, int x, int y)
{
    static GLintPoint corner[2];
    static int numCorners = 0;    // initial value is 0
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        corner[numCorners].x = x;
        corner[numCorners].y = screenHeight - y; //flip y coord
        numCorners++;
    }
}
```

**Screenheight is height of drawing window**
Mouse Interaction Example (continued)

if(numCorners == 2)
{
    // draw rectangle or do whatever you planned to do
    Point3 points[4] = corner[0].x, corner[0].y,
                    corner[1].x, corner[0].y,
                    corner[1].x, corner[1].y,
                    corner[0].x, corner[1].y);

    glDrawArrays(GL_QUADS, 0, 4);

    numCorners == 0;
}
else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    glClear(GL_COLOR_BUFFER_BIT); // clear the window
    glFlush();
}
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 = InitShader( "vshader1.glsl", "fshader1.glsl" );
glUseProgram(program);
```

What’s inside `initShader`??

Next!

**example.cpp**

Main Program

- Vertex shader
  - `vshader1.glsl`
- Fragment Shader
  - `fshader1.glsl`
Coupling Shaders to Application (initShader function)

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

```c
GLuint myProgObj;
myProgObj = glCreateProgram();
```

Create container called **Program Object**

Main Program
Step 2: Read a Shader

- Shaders compiled and added to program object

- 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 Reader Code?

```c
#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;  

GLchar vShaderfile[] = "vshader1.glsl";  
GLchar fShaderfile[] = "fshader1.glsl";  

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

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

Example.cpp

Main Program

Vertex shader
vshader1.glsl

Fragment Shader
fshader1.glsl
Step 3: Adding + Compiling Shaders

Step 4: Link Program

Read shader code strings into shader objects

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

Compile shader objects

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

Attach shader objects to program object

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

Link Program

```c
glLinkProgram(myProgObj);
```

Diagram:
- `example.cpp`
  - `Main Program`
  - `Vertex shader`
    - `vshader1.glsl`
  - `Fragment Shader`
    - `fshader1.glsl`

- 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);
  ```

- 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 of shader variables, each with an index
- Application can get index from table, tie it to application variable
- In application, find location of shader **time** variable in linker table

```c
Glint timeParam;
timeParam = glGetUniformLocation(program, "time");
```

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

```c
glUniform1(timeParam, 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 \( v\text{Position} \)

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

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

Enable vertex array attribute at location of \( v\text{Position} \)
Specify vertex array attribute at location of \( v\text{Position} \)
glVertexAttribPointer

- Vertices are packed as array of values

```
glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Vertices stored in array

```
x y x y x y x y
```

- Padding between consecutive vertices
- Data starts at offset from start of array
- Data not normalized to 0-1 range
- Location of \textit{vPosition} in table of variables
- 2 elements of floats per vertex

Location of \textit{vPosition} in table of variables
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

```c
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 (ivec2, ivec3, ivec4) and boolean (bvec2, bvec3, bvec4)
- **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
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.** Two possibilities
  - **in:** variables copied in
  - **out:** returned values are copied back
- **inout** (deprecated)
- **Example:** vertex shader using **out**

```c
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
  ```c
  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);`
References